INTERDEPENDÊNCIA ENTRE ALOCAÇÃO DE REGISTRADORES E ESCALONAMENTO DE INSTRUÇÕES: ESTUDO SISTEMÁTICO E VERIFICAÇÃO DE SOLUÇÕES

JOÃO FRANCISCO NEIVA DE CARVALHO

INTERDEPENDÊNCIA ENTRE ALOCAÇÃO DE REGISTRADORES E ESCALONAMENTO DE INSTRUÇÕES: ESTUDO SISTEMÁTICO E VERIFICAÇÃO DE SOLUÇÕES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha

Belo Horizonte

Março de 2018

Carvalho, João Francisco Neiva de.

C331i Interdependência entre alocação de registradores e escalonamento de instruções [manuscrito]: estudo sistemático e verificação de soluções / João Francisco Neiva de Carvalho. - 2018.
xxix, 165 f. il.

Orientadora: Mariza Andrade da Silva Bigonha
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.155-165
.
1. Computação – Teses. 2. Compiladores (Computadores). – Teses. 3. Software – Desenvolvimento – Teses. 4. Otimização de código - Teses. I. Bigonha, Mariza Andrade da Silva. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Interdependência entre alocação de registradores e escalonamento de instruções: estudo sistemático e verificação de soluções

JOÃO FRANCISCO NEIVA DE CARVALHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

PROF. CARLOS CAMARÃO DE FIGUEIREDO
Departamento de Ciência da Computação - UFMG

PROFA. KÉCIA ALINE MARQUES FERREIRA
Departamento de Computação - CEFET-MG

PROF. LEONARDO VIEIRA DOS SANTOS REIS
Departamento de Ciência da Computação - UFJF

Belo Horizonte, 28 de março de 2018.

Para Carlos Eduardo e Gabriel, Marina, Rafael e Felipe.

Agradecimentos

Várias foram as pessoas que me apoiaram durante o curso de mestrado. A todas elas, minha sincera e feliz gratidão. De modo especial, agradeço à minha família, Eduarda, Carlos Eduardo e Gabriel, pelas muitas e diferentes formas de apoio e motivação durante este e todos os outros caminhos das nossas vidas. Amo mesmo vocês. Agradeço ao meu pai, poeta, filósofo e meu maior amigo, Carlos de Carvalho, pela presença que me formou, que me inspira e que me acompanha todos os dias. Agradeço à minha mãe, Ana Lúcia, pelo apoio, pelo incentivo, pela disposição de sempre ajudar. Agradeço a Wanderleia Maria de Britto, pela dedicação de sempre e sempre, e pelos cuidados mais constantes. Agradeço aos meus irmãos, Paula e Tiago, e aos meus familiares e amigos, de modo especial, a Maria Cristina, Leila Stein e Helton Cassemiro pelo interesse, pelas perguntas e pela admiração. Minha gratidão aos meus colegas Bruno Luan de Sousa e Marcus Rodrigues de Araújo, com quem elaborei o artigo publicado durante o curso. Minha gratidão também a Mariana Vieira Siqueira de Arantes, minha colega e companheira de trabalho. Meu muitíssimo obrigado à minha orientadora, por quem tenho grande admiração e carinho, professora Mariza Andrade da Silva Bigonha, pelo apoio, dedicação, entendimento e comprometimento durante toda essa jornada. Conseguimos, afinal. Finalmente, é preciso dizer que este trabalho foi financiado pelo povo brasileiro, por meio da Capes. Aos financiadores, meu muito obrigado.

"Na vida, não existe nada a temer, mas a entender."
(Marie Curie)

Resumo

A alocação de registradores e o escalonamento de instruções são duas importantes tarefas realizadas por um compilador. A primeira define quais valores do programa deverão ser alocados nos registradores físicos da máquina e associa esses valores com os registradores físicos disponíveis. A segunda escalona as instruções do programa, podendo movimentá-las ao longo do código, a fim de que a sequência de instruções resultante seja executada mais rapidamente do que a sequência de instruções original. Todavia, essas duas tarefas estão envolvidas em um problema de priorização: se a tarefa de alocação for executada antes da tarefa de escalonamento, falsas dependências poderão surgir entre as instruções do código, prejudicando o escalonamento ótimo. Por outro lado, se a tarefa de escalonamento for realizada antes da tarefa de alocação, os valores do programa podem permanecer vivos por um grande número de instruções, aumentando as chances de derramamentos de valores para a memória. Esse problema de priorização, também conhecido como o *problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções*, ganhou relevância com o surgimento das arquiteturas RISC e com o aumento da importância dos compiladores para a geração de códigos eficientes. O trabalho descrito nesta dissertação considera esse problema em duas vertentes distintas e complementares: uma teórica, realizada por meio de uma revisão sistemática da literatura sobre o tema; e uma prática, efetuada a partir da implementação da abordagem de Pinter no LLVM, uma abordagem que visa reduzir o impacto que a interdependência entre as duas tarefas acarreta sobre a geração de códigos eficientes. Os resultados alcançados, principalmente os resultados práticos, indicam que a interdependência entre as duas tarefas continua sendo um problema relevante e mostram que a implementação realizada foi capaz de gerar códigos mais eficientes, quando comparados com os códigos gerados pelo LLVM e pelo Microsoft Visual Studio.

Palavras-chave: Compiladores, Otimização de Código, Alocação de Registradores, Escalonamento de Instruções.

Abstract

Register allocation and instruction scheduling are two important tasks performed by a compiler. The first defines which program values should be allocated in the machine physical registers and associates those values with the available physical registers. The second schedules the program instructions, moving them along the code, so that the resulting instruction sequence is executed faster than the original instruction sequence. However, these two tasks are involved in a prioritization problem: if the allocation task is executed before the scheduling task, false dependencies may arise between instructions, preventing the optimal scheduling. On the other hand, if the scheduling task is performed prior to the allocation task, program values can remain alive for a large number of instructions, increasing the chances of spilling values into memory. This prioritization problem, also known as the *problem of the interdependence between register allocation and instruction scheduling tasks*, has gained relevance with the emergence of RISC architectures and with the increasing importance of compilers for efficient code generation. The work described in this dissertation takes over this problem in two distinct and complementary aspects: a theoretical one, realized through a systematic literature review; and a practical one, realized by the implementation of Pinter's approach into LLVM, an approach that aims to reduce the impact that the interdependence between the two tasks produces on the generation of efficient code. The results achieved, mainly the practical results, indicate that the interdependence between the two tasks remains a relevant problem and show that the realized implementation was able to generate more efficient codes, when compared to codes generated by LLVM and Microsoft Visual Studio compilers.

Keywords: Compilers, Code Optimization, Register Allocation, Instruction Scheduling.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AR Alocação de Registradores

ARM Advanced RISC Machine

ASIC Application Specific Integrated Circuits

ASIP Application-Specific Instruction-set Processors

CBSoft Congresso Brasileiro de Software

CFG Control Flow Graph

CISC Complex Instruction Set Computer

COMPAQ Compatibility and Quality

CPU Central Processing Unit

CRAIG Combining Register Assignment Interference Graphs

CRISP Combined Register allocation and Instruction Scheduling Problem

CRRA Cooperative Re-scheduling Register Allocation

CSP Code Scheduling for Pipelined processors

CSR Code Scheduling to minimize Registers usage

CSV Comma Separeted Values

DAG Directed Acyclic Graph

DDG Data Dependence Graph

DEC Digital Equipment Corporation

DSP Digital Signal Processing

DSP Digital Signal Processor

EI Escalonamento de Instruções

EPIC Explicitly Parallel Instruction Computing

HDSFG Hierarchical Decorated Signal Flow Graph

HLS High-level Synthesis

HP Hewlett-Packard

IA-64 Itanium Architecture 64-bits

IEEE Institute of Electrical and Electronics Engineers

ILP Integer Linear Programming

IP Cores Intellectual Property Cores

IPS Integrated Prepass Scheduling

IRIS Integrated Register Allocation and Instruction Scheduling

ISA Instruction Set Architecture

JIT Just In Time

LIW Long Instruction Word

LLC LLVM Static Compiler

LSSA Linear Static Single Assigment

MC Machine Code

MDFG Memory-access Data Flow Graph

MIPS Microprocessor without Interlocked Pipeline Stages

MRB Minimum Register Bound

PBQP Partitioned Boolean Quadratic Problem

PDG Program Dependence Graph

PDP Programmed Data Processor

PEOS Performance and Energy Optimal Scheduling

POSIX Portable Operating System Interface

QAP Quadratic Assignment Problem

QP Questões de Pesquisa

RASE Register Allocation with Schedule Estimates

RASER Register Allocation Sensitive Region Scheduling

RCRS Register Constrained Rotation Scheduling

RISC Reduced Instruction Set Computer

RSL Revisão Sistemática da Literatura

SBLP Simpósio Brasileiro de Linguagens de Programação

SPARC Scalable Processar Architecture

SPEC Standard Performance Evaluation Corporation

SSA Static Single Assigment

SSG Scheduler Sensitive Global Register Allocator

UCRT Universal C Run-Time

URM Unlimited Register Machine

VAX Virtual Address Extension

VLIW Very Long Instruction Word

WAR Write-After-Read

Sumário

Capítulo 1

Introdução

O mundo como o conhecemos depende cada vez mais dos computadores. Atualmente, embora isso ainda não seja impossível, é difícil imaginar atividades humanas que não necessitem, de algum modo, ou em algum momento, de alguma ferramenta computacional.

Quaisquer tarefas realizadas por um computador foram previamente definidas em um *software*, escrito em alguma *linguagem de programação*. As linguagens de programação descrevem computações para humanos e máquinas, mas, via de regra, são inteligíveis apenas pelos seres humanos.

Assim, para que a execução de um *software* seja possível, é necessário que ele seja traduzido para um formato adequado, capaz de ser compreendido pelo computador. Genericamente, dizemos que o resultado dessa tradução é a *linguagem de máquina*. Tal linguagem também descreve computações para humanos e máquinas, mas é facilmente inteligível apenas pelos computadores.

A tradução do *software* para a linguagem de máquina é feita por um sistema especializado, denominado *compilador*. Um compilador é, portanto, um sistema capaz de mapear um código escrito em uma determinada linguagem de programação, denominado *programa fonte*, para um código semanticamente equivalente, chamado *programa objeto*.

Geralmente, um compilador pode ser dividido em duas partes: *frontend*, responsável pela análise do programa fonte, e *backend*, responsável pela construção do programa objeto, conforme ilustra a Figura 1.1.

O *frontend* recebe como entrada o programa fonte e, após constatar que ele foi escrito de acordo com as regras da linguagem de programação escolhida, produz como resultado uma representação intermediária desse programa, frequentemente denominada *código intermediário*. O *backend* recebe então o código intermediário gerado e

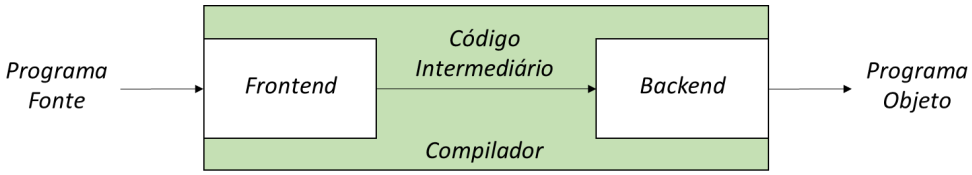


Figura 1.1. Representação de um compilador.

constrói o programa objeto para uma *máquina alvo*[1] específica. Desse modo, enquanto o *frontend* depende da linguagem de programação que foi escolhida, o *backend* depende da arquitetura da máquina que será utilizada.

Na maior parte das situações, desejamos que os computadores executem suas tarefas no menor tempo possível[2]. De modo geral, para um determinado programa fonte e uma determinada máquina alvo, o tempo total de processamento de um *software* será tão menor quanto melhor for o processo de tradução feito pelo compilador. Isto é, quanto melhor o *backend* do compilador souber utilizar as características da arquitetura da máquina alvo, que são capazes de influenciar o tempo de execução do código objeto, melhor será o resultado.

Dentre as tarefas realizadas pelo *backend*, duas são especialmente importantes para o desempenho do programa objeto: a *alocação de registradores* (AR) e o *escalonamento de instruções* (EI). Ambas dependem diretamente da arquitetura da máquina alvo. Via de regra, tanto o desempenho do código gerado, quanto o código em si mesmo, isto é, as instruções que o compõem, são impactados pelo tratamento que se dá a cada uma delas e, em muitos casos, pelo momento em que elas são executadas. Tratam-se de tarefas distintas, que podem ser realizadas pelo compilador em momentos diferentes, o que quase sempre ocorre, mas que interferem entre si.

A alocação de registradores é a tarefa de mapear cada uma das variáveis temporárias utilizadas no código intermediário para um registrador da máquina alvo, a fim de minimizar o número de acessos à memória. O escalonamento de instruções, por sua vez, reordena as instruções do código intermediário, a fim de que elas possam ser executadas em paralelo.

Essas duas tarefas tentam minimizar o tempo de execução do programa, cada uma a seu modo. Contudo, elas estão envolvidas em um problema de priorização, que será apresentado detalhadamente na Seção 2.6. Resumidamente, tal problema

[1]Qualquer máquina construída com a mesma arquitetura considerada no projeto do compilador.

[2]Em algumas situações, outros requisitos de desempenho, como menor utilização de memória, ou menor consumo de energia, podem prevalecer. Nesta dissertação, sempre que nos referirmos ao desempenho de um programa, estaremos falando sobre o seu tempo de execução. Quanto menor o tempo de execução, melhor o desempenho.

pode ser formulado da seguinte maneira: se a alocação de registradores for realizada primeiramente, a solução de mapeamento adotada pode restringir as possibilidades de escalonamento e, por essa razão, prejudicar o desempenho do programa gerado; de outro modo, se o escalonamento de instruções for executado em primeiro lugar, a sequência de instruções escolhida pode acarretar um mapeamento com um número elevado de acessos à memória, também prejudicando o desempenho do programa. Assim, podemos dizer que, exceto para casos particulares, ou muito triviais, as soluções ótimas e independentes de cada uma dessas tarefas são incompatíveis, de modo que torna-se difícil para o compilador utilizá-las em conjunto para gerar o código objeto.

Nesta dissertação de mestrado, apresentamos os resultados da pesquisa que realizamos sobre essas duas tarefas específicas, bem como sobre o problema gerado pela interferência que elas exercem entre si, conhecido como o *problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções*. Além da pesquisa teórica realizada, um algoritmo identificado na literatura, capaz de minimizar os efeitos negativos dessa interdependência, foi implementado e testado em um compilador real. Os testes realizados mostraram que os códigos gerados a partir dessa implementação alcançaram ótimos resultados de desempenho, ratificando as constatações decorrentes da pesquisa teórica, isto é, comprovando que esse campo de pesquisa ainda está aberto a novas ideias e soluções.

1.1 Motivação

O problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções surgiu praticamente em conjunto com as arquiteturas RISC (*Reduced Instruction Set Computer*) na década de 80 [M. Bigonha, 1992]. Desde então, várias soluções para mitigá-lo vêm sendo propostas e ele continua sendo um tema de pesquisa atual, como demonstram alguns estudos [Kri & Feeley, 2004; Kim & Lee, 2010; Philippidis & Shang, 2010; Ivanov, 2010; Huang et al., 2011; Zhang et al., 2011; Kiran et al., 2015; Domagala et al., 2016; Lozano et al., 2012, 2016].

Trata-se de um problema potencialmente relevante para quaisquer arquiteturas que necessitem do compilador para a realização das tarefas de alocação de registradores e escalonamento de instruções[3].

Por fim, é notório que não há consenso na literatura sobre aspectos importantes desse problema, tampouco sobre o verdadeiro impacto que ele exerce sobre a qualidade do código objeto gerado. Assim, apesar dos estudos já realizados, ainda há questões

[3] A Seção 2.3.1.5 abordará em detalhes essa questão.

em aberto, que motivam e justificam a realização de trabalho de pesquisa adicionais, tais como:

1. Em que ordem as tarefas de alocação de registradores e escalonamento de instruções devem ser executadas?
2. A interdependência entre essas tarefas afeta a geração de código para todas as arquiteturas?
3. A ordem de execução dessas tarefas é relevante para a geração de código independentemente da arquitetura?
4. Há alguma ordem de execução preferencial válida para todas, ou para a maioria das situações e arquiteturas?
5. Qual deve ser o nível de comunicação entre as tarefas de alocação e escalonamento para que a compilação do código seja eficiente?
6. Qual deve ser o nível de comunicação entre essas tarefas para a geração de um código eficiente?
7. Em que medida um código ineficiente está associado à falta de comunicação entre essas tarefas?

De modo mais abrangente e indireto, este trabalho pode ser útil na busca de respostas para todas essas questões. De modo direto e específico, este trabalho relaciona-se principalmente às duas últimas.

1.2 Objetivos

Para a pesquisa descrita nesta dissertação, dois objetivos principais foram determinados:

(i) Compreender o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, avaliando seus efeitos sobre a otimização do código e identificando na literatura os principais métodos e abordagens para lidar com ele.

(ii) Dentre os métodos encontrados, identificar o que melhor soluciona o problema e implementá-lo em um compilador real, verificando a sua eficiência por meio de testes e comparações com outros métodos eventualmente existentes no compilador utilizado.

Para atingirmos esses objetivos, definimos os seguintes objetivos específicos:

1. Realizar uma *revisão sistemática da literatura* sobre o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções.

   A finalidade dessa revisão é atender ao primeiro dos dois objetivos principais. É relevante destacar que tal revisão, sendo *sistemática*, presume um caráter metódico e científico, opondo-se a uma mera pesquisa bibliográfica, ou a uma simples compilação de informações. De fato, tal revisão foi realizada conforme os procedimentos definidos por Kitchenham em seus relatórios técnicos, que são referência para esse tipo de trabalho [Kitchenham, 2004].

2. Identificar, a partir da revisão sistemática elaborada, as principais abordagens utilizadas para lidar com o problema da interdependência.

3. Dentre as técnicas identificadas, selecionar aquela que melhor soluciona o problema e implementá-la no LLVM[4].

4. Compilar os *benchmarks* do conjunto SPEC CPU2006 [Henning, 2006] da SPEC (*Standard Performance Evaluation Corporation*) utilizando a técnica implementada, e medir o desempenho dos programas assim gerados.

5. Comparar os desempenhos dos *benchmarks* compilados com a técnica implementada com os desempenhos desses mesmos *benchmarks* compilados com os métodos de alocação e escalonamento existentes no LLVM.

6. Avaliar e justificar os resultados obtidos, identificando aspectos positivos e negativos do método implementado.

1.3 Publicação e Contribuições

A pesquisa descrita nesta dissertação produziu a seguinte publicação: *The Register Allocation and Instruction Scheduling Challenge*, João F. N. Carvalho, Bruno L. Sousa, Marcus R. Araújo, Mariza A. S. Bigonha, *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ACM, doi:10.1145/3125374.3125380, 2017.

[4]Portfólio composto por uma série de tecnologias e ferramentas voltadas para o desenvolvimento de compiladores [Team, 2017]. Nesta dissertação, por simplicidade e clareza, sempre nos referiremos ao LLVM como se ele fosse uma única ferramenta, um único compilador, cujo código fonte pode ser alterado e configurado, permitindo o acréscimo de novas funcionalidades. A Seção 5.1 apresenta mais informações sobre o LLVM.

Além dessa publicação, outras contribuições significativas para o entendimento e para o tratamento do problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções também foram realizadas. Dentre elas pode-se destacar uma implementação que foi capaz de gerar códigos de alto desempenho para a plataforma Windows® da Microsoft. Para um melhor entendimento sobre quais são e como foram alcançadas as contribuições deste trabalho, optamos por apresentá-las de forma mais detalhada no fim desta dissertação, na Seção 8.1.

1.4 Organização do Texto

Além deste capítulo, esta dissertação de mestrado possui outros sete capítulos que estão organizados como se segue.

O Capítulo 2 revisa os principais conceitos e teorias relacionados ao tema pesquisado. Ele também descreve em detalhes o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Trata-se de um capítulo que pode ser útil para os leitores menos familiarizados com a teoria de compiladores.

O Capítulo 3 é dedicado à apresentação da revisão sistemática da literatura sobre o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções e apresenta todo o processo de construção dessa revisão.

Por sua vez, o Capítulo 4 descreve o método escolhido como solução para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, bem como o processo de escolha realizado.

O Capítulo 5 é dedicado ao LLVM e ao conjunto de *benchmarks* SPEC CPU2006, as ferramentas relacionadas, respectivamente, à implementação e aos testes da solução escolhida, apresentada no capítulo anterior.

O Capítulo 6 descreve a implementação que foi feita no LLVM da solução escolhida e indicada no Capítulo 4. As características e estruturas de dados mais relevantes dessa implementação são apresentadas. Como indicado na Seção 1.3, essa implementação utilizou a plataforma Windows® da Microsoft. Esse é um ponto de diferenciação do trabalho, uma vez que desenvolvimentos utilizando ferramentas de software livre, como o LLVM, para essa plataforma não são comumente encontrados, principalmente no meio acadêmico.

O Capítulo 7 apresenta os testes da implementação feita e analisa criticamente os resultados obtidos. Vale adiantar que comparações de desempenho entre compilações com a solução escolhida e compilações com o LLVM e com Visual Studio®, compilador

da própria Microsoft, indicaram vantagens para a solução que foi implementada.

Por fim, o Capítulo 8 conclui esta dissertação de mestrado, detalhando as suas contribuições e indicando possibilidades para trabalhos futuros. Segue-se a ele as referências bibliográficas utilizadas nesta dissertação.

1.4.1 Estrangeirismos

Durante a elaboração desta dissertação, tornou-se importante considerar o fato de que alguns leitores podem se sentir incomodados pela utilização de palavras e expressões estrangeiras ao longo do texto.

De fato, os estrangeirismos têm causado polêmica e têm sido analisados e discutidos há tempos por diversos gramáticos e estudiosos do tema. Alguns os consideram vício literário e defendem que é necessário combatê-los, visando o cuidado com o idioma e o respeito à nacionalidade [Almeida, 1979]. Outros defendem que os estrangeirismos não descaracterizam a língua e sim enriquecem-na por meio da ampliação semântica decorrente do dinamismo cultural, que é forjado pelo *"gênio inventivo do povo"* [Cunha, 1975; Castro, 2002; Valadares, 2014].

Em respeito aos possíveis leitores cujo pensamento se alinha com o primeiro entendimento, cumpre registrar que, durante a elaboração desta dissertação, não houve descuido com a nossa língua portuguesa, tampouco com a nossa cultura.

Os estrangeirismos utilizados neste trabalho não passaram desapercebidos, como se estivéssemos obrigados a nos conformar com a existência deles. As palavras e expressões estrangeiras que nos permitimos utilizar visaram tão somente a precisão na transmissão das informações. Em alguns casos, como ocorre na Seção 3.3.1, são elas títulos de obras e nomes de ferramentas estrangeiras, que preferimos não traduzir por respeito ao artefato original e, principalmente, para minimizar equívocos de entendimento. Em outros casos, que podem ser pinçados ao longo do texto, tratam-se de palavras e expressões presentes em grande parte da literatura existente sobre o tema e que há tempos têm sido utilizadas, se não por todo o povo, ao menos pela maior parte da comunidade científica e acadêmica interessada. É fácil comprovar que muitas dessas palavras, como *software* e *hardware* por exemplo, já foram incorporadas ao nosso vocábulo e já adquiriram significado semântico mais amplo em nossa língua, extrapolando seu estrito significado original [Valadares, 2014].

Finalmente, talvez não seja menos relevante considerar a dificuldade atual de se escrever um texto científico sem utilizar quaisquer palavras estrangeiras e sem comprometer, por esse exato motivo, a clareza e a precisão das informações. Talvez também seja apropriado questionar a utilidade real de se fazer isso atualmente. Tal dificuldade

e, eventualmente, a pouca utilidade que se conclua existir nesse esforço são consequências da globalização e do intercâmbio científico contemporâneo. Fatos inquestionáveis que envolvem o Brasil e, praticamente, todo o mundo e que interferem na produção científica de todos os países, levando, possivelmente, de algum modo e em algum grau, os estrangeirismos à produção científica de todos eles.

Capítulo 2

Referencial Teórico

Apresentamos neste capítulo os principais conceitos teóricos utilizados nesta dissertação. Iniciamos com conceitos fundamentais, para depois abordarmos os temas de Otimização de Código, Geração de Programa, Alocação de Registradores e Escalonamento de Instruções. Terminamos o capítulo descrevendo com mais detalhes o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, uma vez que ele é o tema central deste trabalho.

2.1 Conceitos Fundamentais

Um compilador é um sistema que recebe como entrada um programa fonte, geralmente escrito em uma linguagem de programação de alto nível, e produz como saída um programa objeto que é semanticamente equivalente ao fonte recebido. Normalmente, dividimos o compilador em duas partes: *análise*, também chamada de *frontend*, responsável pela tradução do programa fonte para um código intermediário; e *síntese*, também conhecida como *backend*, responsável pela geração do programa objeto a partir do código intermediário criado.

Para facilitar a geração do programa objeto, o código intermediário criado pelo *frontend* é divido em blocos básicos. Segundo Allen [1970], "*um bloco básico é uma sequência linear de instruções de programa tendo um único ponto de entrada (a primeira instrução executada) e um único ponto de saída (a última instrução executada)*". Assim, dentro de um bloco básico, somente uma única instrução de desvio poderá existir, seja ela condicional ou incondicional.

Normalmente, as instruções que compõem um bloco básico são escritas utilizando-se um *código de três endereços*. Esse código contempla instruções de atribuição com, no máximo, um único operador binário, aritmético ou lógico no lado direito. Instruções

de desvio condicional ou incondicional, instruções com operadores unários, instruções de cópia e chamadas de procedimentos também estão previstas e são comumente consideradas instruções de três endereços [Aho et al., 2006].

Utilizando-se esse código, uma expressão qualquer será traduzida para um conjunto formado por uma ou mais instruções sequenciais de três endereços, cada uma delas possuindo a forma $x := y\ op\ z$, na qual op é um operador, e x, y e z são constantes, endereços de memória, ou variáveis temporárias.

Conforme Aho et al. [2006], a representação de um bloco básico também pode ser feita utilizando-se um *grafo acíclico dirigido*, um DAG (*Directed Acyclic Graph*). Nesse DAG, os vértices que representam as variáveis vivas na saída do bloco, isto é, que representam as variáveis cujos valores poderão ser utilizados em outros blocos básicos, são denominados *vértices de saída*.

Uma vez que um bloco básico é um conjunto de instruções de três endereços, o DAG de um bloco básico pode ser visto como uma extensão natural do DAG utilizado para representar expressões simples. Neste último, as folhas representam os operandos atômicos[1] e os nós internos representam os operadores da expressão. Os filhos de um nó qualquer são sempre os operandos associados ao operador representado pelo nó, e os nós que representam subexpressões comuns, isto é, subexpressões utilizadas mais de uma vez dentro da expressão, não precisam ser repetidos.

Nesses dois DAGs, todas as arestas sempre possuem a mesma direção, ou seja, todas elas apontam dos filhos para os pais, ou, ao contrário, dos pais para os filhos. De fato, essa direção não é relevante para o entendimento desses grafos e, por essa razão, tais arestas quase sempre são representadas graficamente apenas com traços simples e não com setas. O correto modo de leitura de ambos os grafos é de baixo para cima, do nó mais inferior para o que está imediatamente acima dele, sucessivamente, até que a raiz seja alcançada.

Para ilustrar as formas de representação citadas, considere a expressão algébrica $a + b * (a + b + c) + d$. O DAG que representa essa expressão pode ser visto na Figura 2.1 (a). Considere também um código fonte formado somente por essa mesma expressão. As Figuras 2.1 (b) e (c) apresentam, respectivamente, o bloco básico gerado pela tradução desse código para instruções de três endereços, e o DAG que representa esse mesmo bloco básico. Neste DAG, o único vértice de saída é o que possui o rótulo $t4$, pois é ele que representa a única variável que possivelmente precisaria estar viva na saída do bloco básico, ou seja, o resultado final da expressão algébrica.

Após a divisão do código intermediário em blocos básicos, o fluxo de controle

[1] Variáveis ou constantes de tipos de dados simples.

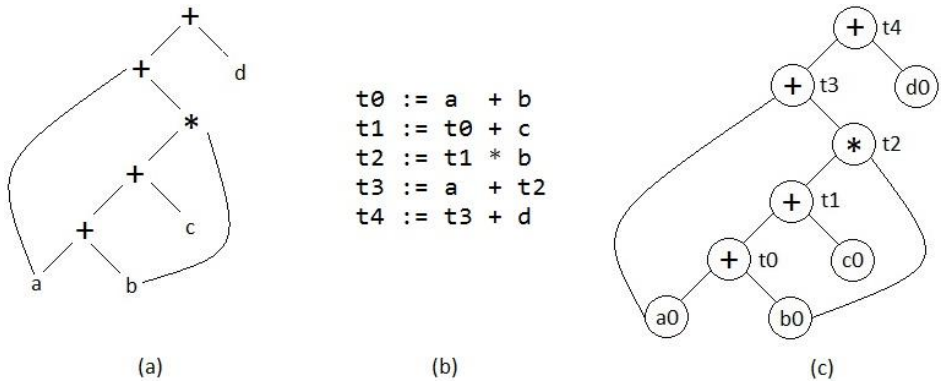


Figura 2.1. DAG para expressão algébrica e representações de um bloco básico: (a) DAG para a expressão algébrica *a + b * (a + b + c) + d*, (b) bloco básico correspondente à expressão e (c) DAG do bloco básico.

do programa é usualmente representado por um grafo dirigido, denominado Grafo de Fluxo de Controle (*CFG - Control Flow Graph*). Os vértices desse grafo representam os blocos básicos e as suas arestas representam os possíveis caminhos do fluxo de execução. Existe uma aresta dirigida que sai de um vértice B para um vértice C desse grafo se e somente se o primeiro comando do bloco básico representado pelo vértice C puder ser executado imediatamente após o último comando do bloco básico representado pelo vértice B. Neste exemplo, dizemos que B é antecessor de C e que C é sucessor de B. Assim, dependendo do fluxo de controle do programa, um bloco básico pode possuir vários sucessores e predecessores, e pode ser ele mesmo o seu próprio sucessor.

Em um CFG, dois blocos especiais são utilizados: o bloco de entrada e o bloco de saída. O primeiro é um bloco que não possui antecessores e que representa o ponto inicial de execução do programa. O segundo representa o ponto de saída do programa e não possui sucessores. Para a representação dos CFGs, qualquer estrutura de dados própria para a representação de grafos pode ser utilizada.

Para a construção do CFG, precisamos primeiramente identificar os blocos básicos do código intermediário. O primeiro passo para essa identificação é encontrar a primeira instrução de cada um desses blocos, que é chamada de *instrução líder*. De acordo com Aho et al. [2006], para cada líder haverá um bloco básico, que será formado pelo próprio líder e por todas as instruções sequenciais existentes até o próximo líder, sem incluí-lo, ou até o fim do código intermediário. Ainda de acordo com Aho et al. [2006], uma instrução será líder se estiver em ao menos uma das seguintes situações:

- se for a primeira instrução do código intermediário;
- se for a instrução de destino de algum desvio condicional ou incondicional;

- se for a instrução imediatamente após um desvio condicional ou incondicional.

A Figura 2.2 ilustra as etapas para a criação do grafo de fluxo de controle de um determinado código fonte que calcula a média aritmética do intervalo de números inteiros $[0, numero]$, sendo *numero* um valor maior ou igual a zero. A partir do código fonte apresentado em (a), o *frontend* do compilador gera o código intermediário indicado em (b), como um conjunto de instruções de três endereços. Neste código, quatro blocos básicos são identificados, blocos *B0* a *B3*, utilizando-se o processo descrito anteriormente. A partir desses blocos e dos caminhos de fluxo de controle existentes no programa, o CFG apresentado em (c) é construído.

O código intermediário é uma representação direta e sem qualquer otimização do DAG que corresponde às instruções do código fonte. As variáveis temporárias *t0* a *t4* são os pseudoregistradores utilizados pelo compilador para a criação das instruções de três endereços.

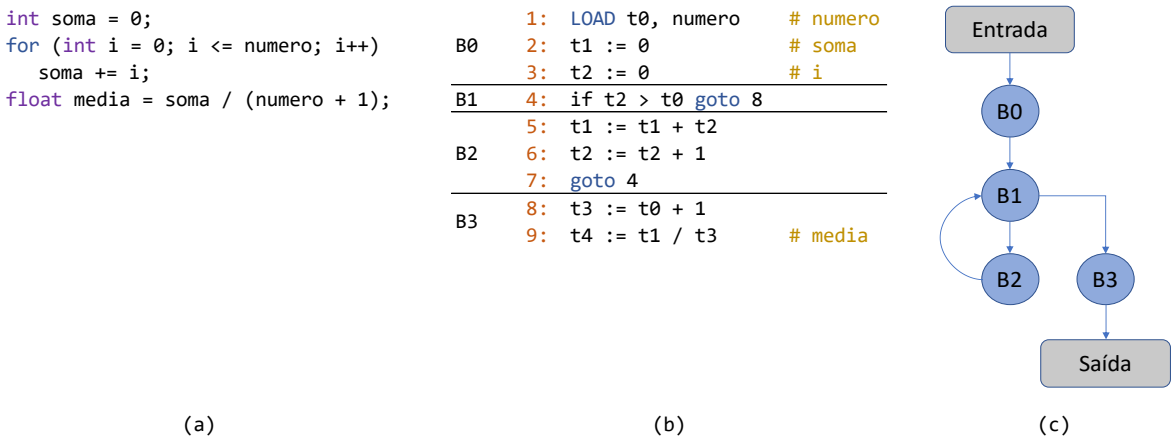


Figura 2.2. Criação do grafo de fluxo de controle: (a) Código fonte; (b) Código intermediário; (c) CFG.

Em um grafo de fluxo de controle, um bloco básico B *domina* um bloco B', se todos os caminhos entre o bloco de entrada e B' passam por B. Por outro lado, B' *pós-domina* B, se todos os caminhos a partir de B para o bloco de saída passam por B'. Como exemplo, todos os blocos básicos do CFG da Figura 2.2 são dominados e pós-dominados pelos blocos *B0* e *B3* respectivamente.

Dois blocos básicos B e B' são considerados *equivalentes por controle*, quando o primeiro domina o segundo, e o segundo pós-domina o primeiro. Sempre que um desses blocos for executado, o outro também será. Como corolário, um desses blocos será executado se e somente se o outro também for executado [Aho et al., 2006].

Outro conceito importante definido a partir dos grafos de fluxo de controle é o conceito de *região*. Essencialmente, uma região é um determinado conjunto de blocos básicos do programa. Formalmente, uma região é um subgrafo $R = (V, E)$ do grafo de fluxo, tal que: ❶ existe um vértice d em V que domina todos os outros vértices em V, isto é, existe um vértice d em V que representa um bloco básico que domina todos os outros blocos representados pelos demais vértices existentes em V; ❷ se algum nó v puder alcançar um nó w pertencente a V, sem passar por d, então v também deve pertencer a V; e, finalmente, ❸ E é o conjunto de arestas que conectam quaisquer pares de vértices que pertencem a V.

Assim, tomando o grafo da Figura 2.2 (c) como exemplo, podemos verificar que o subgrafo formado pelos blocos básicos $B0$ e $B1$ e pela aresta $B0 \rightarrow B1$ é uma região. Por outro lado, não há como definir uma região somente com os blocos básicos $B0$, $B1$ e $B3$, uma vez que o bloco $B2$ pode alcançar os blocos $B1$ e $B3$ e, necessariamente, precisa ser incluído na região.

2.2 Otimização de Código

Conforme ilustrado na Figura 2.3, entre o *frontend* e o gerador de código do *backend* é comum que exista uma etapa de otimização, na qual várias transformações independentes de máquina são realizadas no código intermediário. Tais transformações devem preservar a semântica do código original e fazer com que o programa objeto produzido tenha um desempenho melhor do que teria, caso fosse construído a partir do código intermediário não otimizado.

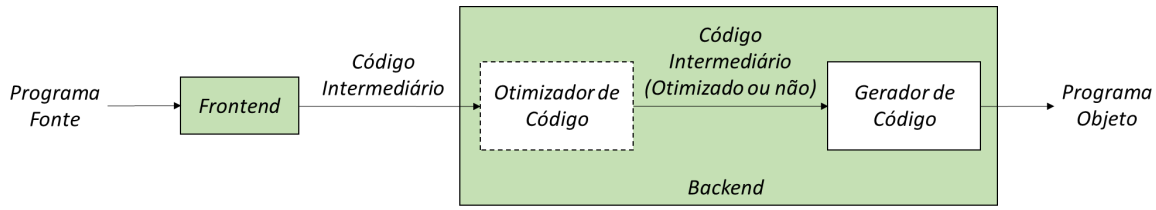


Figura 2.3. Otimizador e Gerador de Código.

As otimizações realizadas dentro de cada bloco básico do código são chamadas de *otimizações locais*, e aquelas que consideram o conjunto de blocos básicos do código, ou um subconjunto deles, examinando como a informação flui entre eles, são chamadas de *otimizações globais* [Torczon & Cooper, 2011; Aho et al., 2006].

Quando um bloco básico é representado por um DAG, várias otimizações locais podem ser realizadas. Exemplos de tais transformações são:

- eliminação de subexpressões comuns locais, ou seja, eliminação de instruções redundantes, que calculam um valor que já foi calculado;
- eliminação de código morto, isto é, eliminação de instruções que calculam valores que nunca são utilizados;
- reordenação de instruções, visando a diminuição do tempo de utilização dos registradores;
- utilização de identidades algébricas, redução de força, propagação de cópia e desdobramento de constantes, a fim de eliminar ou substituir instruções, conforme apresentado em Aho et al. [2006].

Em relação às otimizações globais, a maior parte delas é realizada utilizando-se técnicas de *análise de fluxo de dados*. Essas técnicas possibilitam a coleta e a análise de informações sobre o programa, viabilizando várias transformações de código. Com a análise de fluxo de dados, todas as transformações relacionadas às otimizações locais podem ser realizadas em um nível global, envolvendo vários blocos básicos do programa. Além disso, otimizações dos laços do programa, tais como *movimentação de código* e *eliminação de variáveis de indução* [Aho et al., 2006], também podem ser realizadas.

Uma técnica simples de otimização é conhecida como *otimização de janela*, ou *peephole*. Essa técnica pode ser utilizada para otimizar o código intermediário antes da geração do programa objeto, ou então o próprio programa objeto, após a etapa de geração de código. Ela se baseia na utilização de uma janela deslizante para selecionar um pequeno conjunto de instruções do código. Tal conjunto é então analisado e, sempre que possível, substituído por algum outro que seja menor, ou mais rápido. As instruções apresentadas na janela deslizante podem estar ou não em sequência. Conforme Aho et al. [2006], algumas das transformações que podem ser realizadas no código utilizando-se a técnica *peephole* são: eliminação de instruções redundantes, otimização de fluxo de controle, simplificações algébricas, redução de força e uso de idiomas de máquina, isto é, uso de instruções específicas da arquitetura.

Considerando que um programa pode ser visto como um conjunto de procedimentos, as otimizações também podem ser classificadas como *intraprocedimentais* ou *interprocedimentais*. As primeiras podem ser otimizações locais ou globais, mas sempre estão restritas a um único procedimento. As segundas são sempre otimizações globais, que operam sobre todo o programa, ou sobre mais de um procedimento [Torczon & Cooper, 2011].

Todas as otimizações citadas anteriormente podem ser classificadas como intraprocedimentais. Elas se baseiam apenas na análise do procedimento que está sendo

otimizado. Tais análises são relativamente simples de serem feitas, contudo, são mais imprecisas em certo sentido. Isso porque pressupõem que o procedimento pode gerar quaisquer efeitos no programa, como modificar quaisquer variáveis visíveis, ou gerar exceções que causam o desenrolar de toda a pilha de execução.

Assim, apesar de permitirem a realização de muitas otimizações, as análises intraprocedimentais não são suficientes para realizar certas otimizações do programa. Otimizações envolvendo apontadores e referências para variáveis e procedimentos podem necessitar de análises interprocedimetais. Essas análises atuam em todo o programa, considerando o fluxo de informações dos procedimentos chamadores para os procedimentos chamados e vice-versa. Uma técnica muito utilizada para a realização dessas análises é a expansão em linha (*inlining*) dos procedimentos, pela qual a chamada de um procedimento é substituída pelo seu corpo, ajustando-se adequadamente a passagem de parâmetros e o valor de retorno [Aho et al., 2006].

2.3 Geração do Programa Objeto

Após ter executado a fase de otimização, o compilador inicia a fase de *geração de código*, como mostra a Figura 2.3.

Um gerador de código realiza três principais tarefas: seleção de instruções, alocação de registradores e escalonamento de instruções [Aho et al., 2006]. A seleção de instruções é a tarefa de se escolher as instruções de máquina que são capazes de implementar as instruções do código intermediário. A alocação de registradores determina os valores que devem ser mantidos nos registradores da máquina e em quais registradores cada um desses valores deve ser armazenado. O escalonamento de instruções está relacionado à escolha da ordem de execução das instruções.

Dentre essas três tarefas do gerador de código, a alocação de registradores e o escalonamento de instruções merecem destaque. Essas duas tarefas são as que mais influenciam na eficiência do código gerado. Elas são as mais difíceis de serem realizadas e, além disso, há entre elas uma interdependência, no sentido de que cada uma delas gera uma interferência na outra da seguinte forma: ao privilegiarmos uma solução ótima encontrada para uma delas, geralmente, inviabilizamos a utilização da solução ótima encontrada para a outra. Por estarem no centro desta dissertação, essas duas tarefas serão consideradas de forma mais detalhada nas Seções 2.4 e 2.5 respectivamente.

Compiladores e *arquiteturas de computador* possuem um relacionamento bastante estreito, pois todo programa objeto é gerado visando uma determinada arquitetura. Por essa razão, a Seção 2.3.1 apresenta os principais paradigmas referentes às arquiteturas

de computador, visando, exclusivamente, situar adequadamente entre eles o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções.

2.3.1 Arquiteturas de Computador

A *arquitetura de um computador* está relacionada às principais características que tipificam um computador e que estão disponíveis para um programador. Essas características indicam os recursos e limites que estão presentes na máquina e que terão influência direta sobre a execução dos programas.

As características definidas pela arquitetura influenciam as implementações do computador. De fato, tais implementações decorrem da arquitetura, mas não são descritas por ela. Elas estão associadas à *organização do computador* e não podem ser vistas por um programador. Em resumo, enquanto a arquitetura refere-se aos atributos da máquina que podem ser vistos e utilizados pelos programadores, a organização refere-se aos mecanismos e dispositivos utilizados para a implementação física desses atributos [Stallings, 2010].

A distinção entre as definições de arquitetura e organização não é tema pacífico na literatura. Alguns autores defendem a ideia de que esses dois conceitos possuem essencialmente o mesmo significado em termos práticos [Tanenbaum, 2005]. Outros afirmam que há um consenso sobre o alcance de cada um deles [Stallings, 2010]. Seja como for, a distinção parece ser bastante útil para o entendimento de questões relativamente importantes sobre o assunto.

Uma dessas questões está relacionada à definição de *família* de processadores. Os modelos da família possuem todos a mesma arquitetura e diferem entre si somente em termos de organização. A arquitetura é comum aos diferentes modelos e mantém a compatibilidade entre eles, ao passo que as diferenças de organização que eles possuem os individualizam, sejam elas relativas ao desempenho, ou a alguma outra característica.

Uma segunda questão refere-se ao crescente aumento de velocidade dos processadores. Desde a década de 1970, a velocidade de processamento das máquinas tem aumentado de forma praticamente exponencial. Em parte, conforme previsto pela Lei de Moore [Moore, 1998; Brock, 2006], esse ganho está relacionado à miniaturização dos componentes utilizados nas pastilhas de processamento. Todavia, ele se deve, sobretudo, às mudanças realizadas na *organização dos processadores*, tais como a utilização intensiva de *pipelines* e o uso de técnicas de execução paralela e especulativa [Stallings, 2010].

Um compilador lida somente com a arquitetura da máquina. Com efeito, a grosso

modo, conhecendo o conjunto de instruções (*Instruction Set Architecture - ISA*) e também a forma de acesso aos registradores e à memória do computador, o gerador de código é capaz de realizar a sua função. Entretanto, no contexto deste trabalho, certos aspectos organizacionais precisarão ser considerados, pois produzem efeitos que se sobrepõem aos resultados de determinadas tarefas do compilador.

Os principais paradigmas relativos às arquiteturas de computador continuam sendo as clássicas arquiteturas CISC (*Complex Instruction Set Computer*), RISC (*Reduced Instruction Set Computer*) e superescalar. Há décadas, essas arquiteturas vêm sendo utilizadas e suas características influenciaram e podem ser encontradas em uma incontável variedade de microprocessadores. Outras arquiteturas totalmente diferentes também foram especificadas e implementadas, mas nenhuma delas teve o mesmo alcance.

2.3.1.1 Arquiteturas CISC

A década de 1970 foi marcada pelo nascimento dos microprocessadores, pelo nascimento dos microcomputadores e pela expansão da arquitetura CISC.

Nessa década, a DEC (*Digital Equipment Corporation*) dominava o mercado de minicomputadores com o PDP-8 (*Programmed Data Processor 8*), que foi lançado em 1965 e vendeu 50 mil unidades. Em 1970, ela consolidou sua liderança nesse mercado com o lançamento do PDP-11, um minicomputador de 16-*bits*, que alcançou grande sucesso, principalmente nas universidades. Oito anos depois, em 1978, a DEC lançou o VAX (*Virtual Address eXtension*), o primeiro minicomputador de 32-*bits*. Os minicomputadores eram equipamentos projetados para serem menos complexos do que os *mainframes*[2], mas para também serem utilizados por múltiplos usuários por meio do uso de terminais [Stallings, 2010].

Paralelamente a isso, em 1971, houve o lançamento do Intel 4004, o primeiro microprocessador do mundo, a primeira pastilha a conter todos os componentes de uma CPU (*Central Processing Unit*). Apenas três anos mais tarde, a Intel introduziu o 8080 de 8-*bits*, o primeiro microprocessador de propósito geral projetado para ser a CPU de um microcomputador. Esse microprocessador foi utilizado no primeiro computador pessoal lançado em 1975, chamado Altair. Quatro anos mais tarde, foi lançado o 8086 de 16-*bits*, primeiro microprocessador de propósito geral da Intel, e uma variante dele, o Intel 8088, foi utilizada no primeiro computador pessoal da IBM, o IBM PC, cujas

[2]Computadores de grande porte e altíssimo custo, que dominaram as décadas de 1950 e 1960. Utilizados por grandes corporações para o processamento de grandes volumes de dados e aplicações críticas. Podiam ser acessados por meio de terminais específicos e atender centenas de usuários simultaneamente. O mercado de *mainframes* tem sido dominado pela IBM até os dias atuais.

primeiras versões usaram como sistema operacional o MS-DOS da Microsoft. Esse microcomputador consolidou o sucesso da Intel no segmento de microprocessadores e foi o ponto de partida para o sucesso da Microsoft nos anos posteriores.

Logicamente, esses lançamentos continuaram e, nos anos seguintes, a Intel lançou o 80286 em 1982, que podia trabalhar com uma memória total de até 16 Mbytes, e o 80386 em 1985, seu primeiro microprocessador de 32-*bits*, considerado por alguns a essência dos futuros processadores Pentium [Tanenbaum, 2005].

Em termos de arquitetura, todos os lançamentos desse período seguiram o mesmo paradigma. Acreditava-se que o conjunto de instruções de máquina deveria oferecer um suporte mais adequado às operações e estruturas de dados utilizadas nas linguagens de programação de alto nível, a fim de tornar a programação em linguagem de montagem (linguagem *assembly*) mais fácil, simplificar os compiladores e melhorar o desempenho dos programas. Assim, paulatinamente, lançamento após lançamento, os conjuntos de instruções desenvolvidos mostravam-se cada vez maiores e incluíam instruções cada vez mais complexas [Aho et al., 2006; Stallings, 2010].

Por essa razão, essas arquiteturas passaram a ser chamadas CISC (*Complex Instruction Set Computer*). Em suas especificações, uma única instrução pode realizar um conjunto complexo de operações, e uma determinada operação pode ser realizada por diversas instruções de máquina diferentes. Além disso, seus conjuntos de instruções também consideram modos complexos de endereçamento de memória e de chamada de procedimentos.

O desempenho de um programa pode ser calculado pela equação 2.1. Por essa equação, o desempenho, entendido aqui como a velocidade de processamento do programa, ou o inverso do tempo gasto para processar o programa, pode ser determinado pelo inverso da multiplicação dos seguintes três fatores: ❶ o tempo de processamento de cada um dos ciclos do processador, ❷ o número de ciclos necessários para o processamento de cada uma das instruções do programa e ❸ o número de instruções que o programa possui. Assim, utilizando instruções mais complexas, capazes de realizar de uma só vez os comandos da linguagem de programação, as arquiteturas CISC buscam maximizar o desempenho do código minimizando o número de instruções por programa, em detrimento do número de ciclos por instrução [Chen et al., 2000].

$$\frac{1}{desempenho} = \frac{tempo}{programa} = \frac{tempo}{ciclo} \times \frac{ciclo}{instrução} \times \frac{instrução}{programa} \tag{2.1}$$

Devido à complexidade das instruções e à maior dificuldade de implementá-las fisicamente na máquina, é comum que as arquiteturas CISC utilizem uma unidade de controle microprogramada, em vez de uma unidade de controle implementada em

hardware. O fato de serem programadas as torna mais lentas, mas mais flexíveis e fáceis de serem projetadas. Elas são programadas com uma linguagem de microprogramação e seus microprogramas são conhecidos como *firmwares* [Stallings, 2010].

Resumidamente, as arquiteturas CISC possuem as seguintes características:

- definem um grande número de instruções de máquina;
- utilizam instruções complexas, capazes de realizar operações intricadas;
- suas instruções podem trabalhar diretamente com a memória do computador e utilizam modos complexos de endereçamento e de chamada de procedimentos;
- suas instruções possuem tamanho variável;
- geralmente, possuem poucos registradores de uso geral;
- suas unidades de controle são microprogramadas, implementadas por *firmwares*.

2.3.1.2 Arquiteturas RISC

À medida que a arquitetura CISC proliferava, algumas das suas características começaram a ser questionadas, e os ganhos previstos com a sua utilização começaram a ser avaliados. Com efeito, instruções de máquina complexas não eram afinal tão fáceis de serem utilizadas e não simplificavam o processo de compilação como se imaginara. Isso porque identificar a instrução de máquina que melhor se ajusta a uma determinada construção de alto nível não é uma tarefa trivial. Na prática, os compiladores acabavam privilegiando as instruções mais simples e quase nunca utilizavam as instruções mais complexas [Tanenbaum, 2005; Stallings, 2010].

Além disso, do ponto de vista do desempenho, o tamanho variável das instruções dificultava o processo de decodificação e fazia com que o número de ciclos necessários para se executar instruções de um mesmo tipo fosse bastante inconstante. Finalmente, a utilização de um conjunto de instruções complexas dificultava e tornava mais oneroso o projeto do processador, acarretando também a utilização de unidades de controle mais lentas, por serem microprogramadas [Ricarte, 1999; Dandamudi, 2005].

Sob esse contexto, em 1980, uma equipe da Universidade da Califórnia de Berkeley iniciou a construção de um microprocessador com arquitetura significativamente diferente da arquitetura CISC. Muitos dos conceitos utilizados nesse novo microprocessador também estavam sendo aplicados no projeto do minicomputador IBM 801 [Cocke & Markstein, 2000; Radin, 1982], que se iniciara em 1975.

Como resultado dos trabalhos da equipe de Berkeley, em 1982, o microprocessador RISC I foi anunciado e, no ano seguinte, foi feito o lançamento do RISC II [Patterson

& Sequin, 1982]. Nascia, dessa maneira, a arquitetura RISC (*Reduced Instruction Set Computer*), com o expresso objetivo de especificar um conjunto de instruções mais simples e não necessariamente com menos instruções [Dandamudi, 2005].

Após os lançamentos que iniciaram o movimento das máquinas RISC, pesquisadores da Universidade de Stanford também concluíram um projeto RISC em 1983. Esse projeto, denominado MIPS (*Microprocessor without Interlocked Pipeline Stages*) [Hennessy, 1984], foi a base para o primeiro processador RISC vendido comercialmente. Tal processador recebeu o nome de MIPS e foi fabricado pela MIPS Computer Systems, que foi fundada por alguns dos pesquisadores que participaram do projeto em Stanford. Atualmente, a empresa passou a se chamar MIPS Technologies e continua desenvolvendo processadores RISC, comercializando-os como *IP cores* (*Intellectual Property cores*), que podem ser licenciados para empresas e aplicações diversas.

No início de 1986, devido ao sucesso do projeto do IBM 801, a IBM lançou o *RT Personal Computer*, RT PC, sua primeira estação de trabalho RISC.

Em meados de 1987, a Sun Microsystems lançou o SPARC (*Scalable PRocessar ARChitecture*), um processador RISC baseado nos projetos RISC I e RISC II desenvolvidos em Berkeley. Em 1989, a arquitetura SPARC tornou-se uma arquitetura aberta, não proprietária, gerenciada pela SPARC International, uma organização de diversas empresas, sem fins lucrativos. Os processadores passaram a ser produzidos pelos membros dessa organização, e o objetivo dessa estratégia era disseminar a arquitetura. Em 2010, a Sun Microsystems foi comprada pela Oracle Corporation e os processadores passaram a ser utilizados e comercializados nos servidores da empresa. Em setembro de 2017, após o anúncio do servidor Oracle Fujitsu SPARC M12-2, a Oracle anunciou o fim da utilização dos processadores SPARC e, atualmente, a Fujitsu é o único membro da SPARC International.

Finalmente, em 1986, a DEC lançou sua máquina RISC chamada DEC Alpha, o primeiro computador a utilizar um processador com arquitetura de 64-*bits*. Os demais computadores com essa arquitetura vieram a ser lançados somente após uma década praticamente [Tanenbaum, 2005]. Apesar da inovação, a DEC foi adquirida pela COMPAQ (*COMPatibility And Quality*) Computer Corporation em 1998, e esta foi adquirida pela HP (*Hewlett-Packard*) em 2002. Atualmente, alguns produtos com a marca Compaq são comercializados pela HP, mas, na realidade, todos os produtos DEC e Compaq deixaram de ser produzidos.

Apesar das particularidades, todas as implementações citadas anteriormente apresentaram características em comum. Tais características definem as arquiteturas RISC e são as seguintes:

- o conjunto das instruções de máquina é reduzido;
- as instruções são simples e codificadas em palavras de tamanho e formato fixo. Os campos da instrução têm posições fixas, especialmente o do código da operação;
- os modos de endereçamento são simples e poucos em quantidade;
- o acesso à memória é feito exclusivamente por instruções de carga e armazenamento (*load* e *store*);
- as operações lógicas e aritméticas são realizadas somente entre registradores, tipicamente utilizando instruções de três endereços;
- o número de registradores de propósito geral é elevado, a fim de diminuir a necessidade de acesso à memória;
- podem utilizar janelas de registradores, limitando o número de registradores visíveis por procedimento;
- a realização de tarefas complexas, como a alocação de registradores e o escalonamento de instruções, é deixada a cargo dos compiladores;
- executam uma instrução por *ciclo de máquina*[3], possibilitando o alcance de altas taxas de execução;
- as unidades de controle são implementadas em *hardware*;
- utilizam *pipelines*[4] para acelerar a execução das instruções.

Com as arquiteturas RISC, os compiladores passaram a assumir um papel mais importante no desenvolvimento das arquiteturas de computadores. Antes delas, eles eram desenvolvidos somente após as máquinas terem sido construídas. Depois delas e ainda hoje, os compiladores são concebidos durante o projeto dos processadores, a fim de auxiliar na avaliação dos recursos propostos na arquitetura [Aho et al., 2006].

Diversos estudos comparativos [Patterson & Sequin, 1982; Heath, 1984; Katevenis, 1985] mostraram que os códigos gerados para o processador RISC I eram, em média, 11% maiores do que os códigos gerados para máquinas CISC. Esses mesmos estudos apontaram que essa diferença não necessariamente representaria um problema, pois, devido ao fato das instruções RISC serem mais simples, utilizando, na maior parte dos casos, referências para registradores em vez de referências para a memória, o maior

[3]Um ciclo de máquina é definido como o tempo necessário para buscar dois operandos em registradores, executar uma operação lógica ou aritmética e armazenar o resultado em um registrador.

[4]Estruturas capazes de processar instruções em uma série de passos sequenciais. Cada um deles é executado em um único ciclo de máquina e é responsável por realizar uma das etapas do processamento total [Ricarte, 1999; Aho et al., 2006]. Pela importância que possuem, os *pipelines* serão explicados de modo mais detalhado ainda nesta seção.

número de instruções poderia ser compensado pelo menor número de *bits* necessários para armazená-las.

Seja como for, considerando novamente a Equação 2.1, é importante ressaltar que as arquiteturas RISC buscam maximizar o desempenho dos programas minimizando o número de ciclos por instrução, em detrimento do número de instruções por programa.

A utilização das *janelas de registradores* é uma novidade das arquiteturas RISC para solucionar possíveis impactos causados pelo *chaveamento de contexto*[5] durante a execução de um programa. Com efeito, as operações necessárias para salvar e restaurar os valores dos registradores, a cada transição de procedimento, são responsáveis por um tempo razoável de processamento. Com o elevado número de registradores da máquina, caso todos eles estivessem disponíveis para todos os procedimentos, o desempenho do programa poderia ser significativamente impactado. Com as janelas de registradores, cada procedimento tem acesso somente a uma parte dos registradores, diminuindo o tempo de chaveamento. Como exemplo, podemos considerar a arquitetura RISC II que possuía um total de 138 registradores, mas disponibilizava aos procedimentos janelas com 22 registradores apenas [Ricarte, 1999].

Uma característica muito importante das arquiteturas RISC é a utilização de *pipelines*. O conceito de *pipeline* remete a uma linha de montagem, na qual a construção de um produto é divida em etapas, cada uma delas responsável pela criação de uma das partes do produto.

Sendo assim, *pipelining* é uma técnica para acelerar a execução de quaisquer tarefas que podem ser divididas em tarefas menores, possíveis de serem executadas separadamente e de forma sequencial. Nas arquiteturas RISC, ela é utilizada, principalmente, para o processamento de instruções e para a realização de operações aritméticas. A Figura 2.4 ilustra um *pipeline* de instruções formado por k estágios. O processamento de uma instrução é iniciado no *Estágio 1* do *pipeline* e finalizado no *Estágio k*. Todos os estágios realizam suas tarefas em um ciclo de máquina, dessa forma, o processamento de qualquer instrução será *finalizado* após k ciclos de máquina.

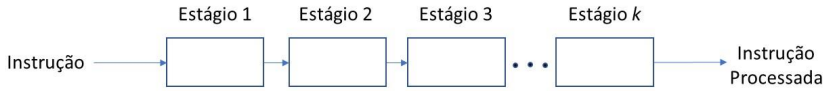


Figura 2.4. *Pipeline* de instruções com k estágios.

Devido à sua eficiência, a técnica de *pipelining* vem sendo utilizada extensamente não apenas nas máquinas RISC, mas também em todas as arquiteturas pós-RISC,

[5]Mudanças no fluxo de execução do código, que implicam operações de salvamento e restauração dos valores dos registradores da máquina.

inclusive nas que são utilizadas nos processadores atuais [Ricarte, 1999].

A Figura 2.5 ilustra o funcionamento do *pipeline* de instrução apresentado anteriormente. Uma vez que o *pipeline* possui k estágios, o processamento da instrução I_1, a primeira instrução, será finalizado no ciclo k. O processamento da instrução I_2, a segunda instrução, será finalizado no ciclo $k+1$ e assim por diante.

A vantagem da técnica de *pipelining* é poder iniciar o processamento de uma instrução antes do processamento da instrução anterior ter sido completamente finalizado. Assim, como ilustra a Figura 2.5, no ciclo k, por exemplo, o processamento da instrução I_1 está sendo finalizado, contudo, o processamento das outras $k-1$ instruções posteriores já foi iniciado. Cada uma delas já está sendo processada em um dos estágios do *pipeline*.

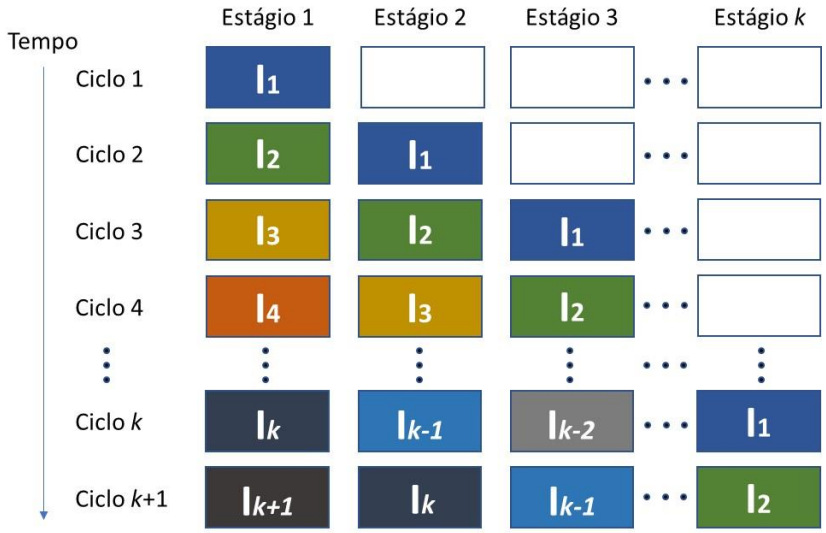


Figura 2.5. Funcionamento de um *pipeline* de instruções com k estágios.

Supondo que n instruções serão processadas por esse *pipeline*, o tempo total (T_p) para o processamento de todas elas será $T_p = kt + (n-1)t$, onde t é o tempo do ciclo de máquina. Isso porque o processamento da primeira instrução será finalizado após kt unidades de tempo, mas, após isso, a cada novo ciclo de máquina, isto é, a cada t unidades de tempo, o processamento de uma das $n-1$ instruções posteriores será finalizado.

Caso essas mesmas n instruções fossem processadas sequencialmente por um único módulo que realizasse as mesmas k etapas, o tempo total de processamento (T_s) seria $T_s = nkt$, uma vez que cada instrução seria processada em kt unidades de tempo, e o processamento de uma instrução qualquer só poderia ser iniciado após o término do processamento da instrução anterior.

A utilização dos *pipelines* é vantajosa porque o número de instruções de máquina que devem ser processadas é quase sempre muito grande. Ou seja, ganha-se em função do volume de instruções. Considerando-se o processamento de apenas uma única instrução, contudo, o tempo de processamento utilizando o *pipeline* é ligeiramente maior do que o tempo teórico, ou seja, ligeiramente maior do que o tempo obtido com o processamento sequencial feito por um único módulo. Isso ocorre devido aos *overheads* intrínsecos ao funcionamento do *pipeline*, sendo o principal deles a transferência de dados entre os diferentes estágios. Logicamente, à medida que o número de estágios aumenta, essa sobrecarga também aumenta, prejudicando o desempenho.

A arquitetura RISC representa uma das mais importantes inovações em termos de arquitetura e organização de computadores. Ela influenciou de modo significativo o projeto de várias outras arquiteturas e microprocessadores posteriores, principalmente em função da técnica de *pipelining*.

Mesmo os processadores CISC tiveram suas arquiteturas alteradas em função da técnica de *pipelining*. Os microprocessadores Intel, cujas arquiteturas talvez ainda sejam o melhor exemplo de arquiteturas CISC na atualidade, foram todos modificados em função dessa técnica. Todos eles, desde o modelo 80486, lançado em 1989, até os mais modernos, utilizam *pipelines*.

Na década de 1980, quando a arquitetura RISC surgiu, várias comparações entre ela e a arquitetura CISC foram feitas. Uma grande polêmica foi criada sobre qual das duas seria a melhor. Diversos artigos científicos foram elaborados, dando conta das vantagens da arquitetura RISC e denunciando os falsos ganhos fornecidos pelas arquiteturas CISC [Hennessy et al., 1982; Patterson & Piepho, 1982]. Alguns outros estudos buscaram identificar e analisar casos em que a utilização de arquiteturas CISC traria ganhos de desempenho [Colwell et al., 1983].

É fácil constatar que muitos estudos recentes ainda discutem esse tema. Contudo, para alguns pesquisadores [Stallings, 2010; Tanenbaum, 2005] esse debate não possui mais sentido. Apesar de alguns fabricantes, como MIPS Technologies e ARM (*Advanced RISC Machine*) ainda produzirem microprocessadores *RISC puros*, grande parte da indústria do setor parece nunca ter se engajado nesse debate, de tal modo que, ao longo dos anos, a evolução dos microprocessadores tem ocorrido com a utilização de tecnologias diversas, originalmente associadas a paradigmas distintos, com o expresso intuito de dominação do mercado por meio da oferta de equipamentos cada vez mais rápidos, ou mais baratos. Com o desenvolvimento tecnológico, tanto os processadores RISC tornaram-se mais complexos, quanto os processadores CISC incorporaram características RISC.

2.3.1.3 Arquitetura Superescalar

Em 1987, Agerwala & Cocke publicaram um relatório descrevendo os resultados do projeto de pesquisa que desenvolveram nos laboratórios da IBM e propondo *"abordagens de compilação, arquitetura e organização de máquina capazes de usar eficientemente os recursos do hardware e alcançar velocidades de execução muito altas"*. O objetivo do projeto era melhorar o desempenho de aplicações que trabalham com operações complexas, tais como operações aritméticas de ponto flutuante. Segundo os autores, o desempenho dessas aplicações seria melhor, se as funções utilizadas por elas estivessem implementadas diretamente no *hardware*. As máquinas com a arquitetura e a organização propostas foram chamadas de *processadores superescalares* [Agerwala & Cocke, 1987].

Processadores superescalares são aqueles que possuem *pipelines de múltiplas instruções*, também chamados de *pipelines superescalares*, que replicam cada um dos estágios de um *pipeline*, de modo que mais de uma instrução pode ser executada no mesmo estágio simultaneamente [Stallings, 2010]. Assim, uma das mais importantes características da arquitetura superescalar é a capacidade de iniciar mais de uma instrução no mesmo ciclo de máquina.

Além disso, os processadores superescalares podem explorar o paralelismo no nível de instrução, isto é, as possibilidades de executar paralelamente as instruções do programa. Para isso, dentre as instruções que estão sendo processadas no *pipeline superescalar*, os processadores identificam e executam paralelamente aquelas que são independentes umas das outras. Caso haja falsas dependências entre algumas instruções, os processadores podem eliminá-las, utilizando técnicas de renomeação de registradores [Stallings, 2010].

A Figura 2.6 apresenta o esquema de um processador superescalar formado por um *pipeline superescalar* capaz de buscar e identificar duas instruções de máquina simultaneamente. Basicamente, o processador desse exemplo é formado por dois *pipelines* com quatro estágios de processamento: busca, decodificação, execução e armazenamento. O estágio de execução possui quatro unidades funcionais[6] que são compartilhadas e utilizadas pelos dois *pipelines*.

Embora a tecnologia superescalar possa ser aplicada mais facilmente às arquiteturas RISC, ela também pode ser utilizada em arquiteturas CISC. Assim, devido à sua grande eficiência, ela tem sido utilizada no projeto de todos os microprocessadores que necessitam de alto desempenho.

[6]Unidades que interpretam e executam as instruções. Tais unidades podem ser quaisquer unidades lógicas e aritméticas por exemplo.

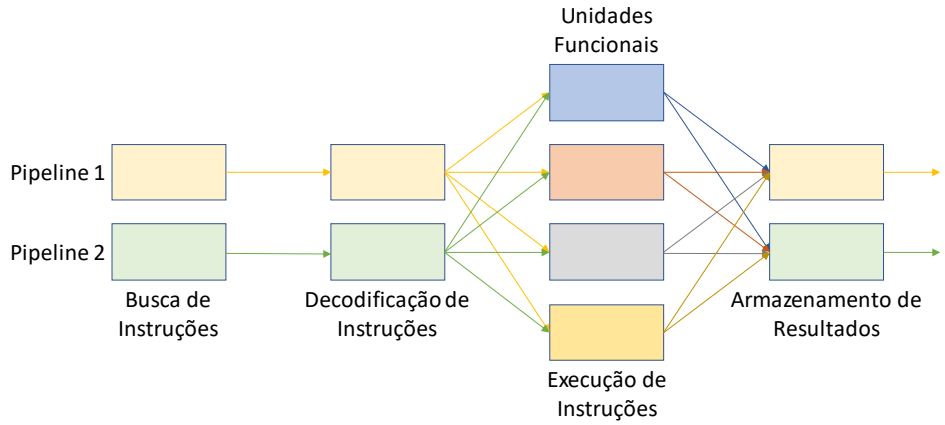


Figura 2.6. Esquema de um processador superescalar.

Desde o processador Pentium, lançado em 1993, a Intel tem incorporado tecnologias superescalares em seus microprocessadores. Atualmente, os processadores Intel, ainda exemplares da arquitetura CISC, utilizam núcleos RISC e princípios de projeto da arquitetura superescalar [Stallings, 2010].

A IBM, por sua vez, tem utilizado a tecnologia superescalar desde o lançamento do IBM RISC Sistema 6000 em 1990, uma máquina RISC superescalar, cuja arquitetura passou a ser chamada POWER naquele mesmo ano. A arquitetura POWER foi a base para a arquitetura PowerPC, criada em 1991, após a associação entre IBM, Motorola e Apple [Stallings, 2010]. Os microprocessadores PowerPC possuem, portanto, uma arquitetura RISC superescalar e foram usados com exclusividade nos computadores da Apple até 2005, quando a empresa anunciou que passaria a utilizar processadores Intel.

Finalmente, a partir do projeto do UltraSPARC II, lançado em 1997, a abordagem superescalar passou a ser utilizada nos processadores da Sun Microsystems. No contexto deste trabalho, é interessante citar esse processador, pois ele foi o sistema utilizado como referência para os *benchmarks* da SPEC CPU2006 [Corporation, 2017b].

O propósito da abordagem superescalar é a exploração do paralelismo de instruções. Para cumprir esse propósito o processador precisa identificar as possibilidades de paralelismo e coordenar todo o processamento das instruções ao longo dos estágios do *pipeline superescalar*.

Nesse sentido, uma questão especialmente importante refere-se ao processo de iniciar a execução das instruções nas unidades funcionais. Na iniciação das instruções, o processador pode escolher a instrução a ser executada, a fim de favorecer o paralelismo, fazendo com que a ordem de execução das instruções seja diferente daquela que existe no programa. Para fazer isso, o processador analisa algumas instruções à frente do ponto de execução corrente e escolhe dentre as instruções possíveis aquela que intensificará o

processamento paralelo. Qualquer instrução pode ser escolhida, desde que o resultado final do processamento de uma sequência de instruções seja igual ao que seria obtido caso a sequência de execução não tivesse sido alterada.

As possíveis *políticas de iniciação de instruções* [Johnson, 1991] para máquinas superescalares são as que se seguem. Uma descrição detalhada de cada uma delas pode ser vista em Stallings [2010]:

- iniciação em ordem, com terminação em ordem;
- iniciação em ordem, com terminação fora de ordem;
- iniciação fora de ordem, com terminação fora de ordem.

Resumidamente, o processo de execução de um programa em uma máquina superescalar é feito conforme os seguintes passos:

1. O processo de busca de instruções recupera as instruções do programa e constrói dinamicamente um fluxo de instruções.
2. O processador examina esse fluxo, verificando e eliminando falsas dependências com a utilização de registradores de nomeação.
3. As instruções são então armazenadas em um janela de execução e são ordenadas de acordo com suas dependências de dados verdadeiras.
4. O processador inicia as instruções existentes na janela de execução, obedecendo a ordem existente.
5. Após executadas, o processo de *confirmação de instrução* é realizado: as instruções são reordenadas conforme a ordem existente no programa e os seus resultados são registrados.

É essencial registrar que a característica de iniciação fora de ordem é muito relevante, tanto do ponto de vista do desempenho dos processadores superescalares, quanto do ponto de vista deste trabalho. De fato, a iniciação fora de ordem representa um escalonamento dinâmico de instruções, feito pelo próprio processador. Na prática, essa política de iniciação retira do compilador e traz para o processador a responsabilidade de definir um escalonamento ótimo de instruções.

2.3.1.4 Outros Paradigmas de Arquitetura

Inúmeras outras arquiteturas foram projetadas e implementadas, mas nenhuma delas teve o mesmo alcance e influência das arquiteturas CISC, RISC e Superescalar. Dentre essas outras arquiteturas, destacam-se as duas seguintes:

- *Superpipelines* - A técnica consiste de um *pipeline* de instruções com um elevado número de estágios muito pequenos, que podem ser executados em um tempo menor do que um ciclo de relógio [Stallings, 2010]. Assim, adotando-se um relógio interno que faça o ciclo de máquina ser menor do que o ciclo de relógio, mais de um estágio do *pipeline* será executado a cada ciclo de relógio, fazendo com que um número grande de instruções esteja no *pipeline* ao mesmo tempo.

  Assim como ocorre em um *pipeline*, em uma superpipeline há uma sobrecarga em função da transfrência de dados entre os estágios. Como exemplo de uma máquina com arquitetura superpipeline, destacava-se o MIPS R4000.

- *VLIW (Very Long Instruction Word)* - A arquitetura VLIW, ou *palavras de instrução muito longas*, é uma alternativa às arquiteturas superescalares para a realização do paralelismo em nível de instrução [Tanenbaum, 2005]. Assim como a arquitetura superescalar, ela também é capaz de iniciar mais de uma instrução por ciclo de máquina e também trabalha com múltiplas unidades funcionais. A diferença entre elas é que a arquitetura VLIW utiliza palavras muito longas para codificar suas instruções. Em um única palavra, mais de uma instrução pode ser codificada. Essas arquiteturas contam com os compiladores para definir os conjuntos de instruções que deverão ser colocados em cada palavra. Dessa forma, o processador não precisa analisar o fluxo de instruções, a fim de eliminar falsas dependências e encontrar a melhor sequência paralelizável.

  Um dos representantes de maior destaque dessa arquitetura foram os processadores Itanium produzidos pela Intel. A arquitetura desses processadores chamava-se IA-64 (*Itanium Architecture 64-bits*) e foi projetada pela Intel e pela HP visando melhorar o desempenho de servidores. Essa arquitetura baseava-se em um conjunto de conceitos básicos, que foram chamados de EPIC (*Explicitly Parallel Instruction Computing*). Dentre esses conceitos estava a utilização de palavras longas (*Long Instruction Word - LIW*) ou muito longas. O primeiro processador Itanium foi lançado em 2001, e a Intel anunciou em maio de 2017 o fim da produção desses processadores.

2.3.1.5 Situando o Problema da Interdependência entre as Tarefas de Alocação de Registradores e Escalonamento de Instruções em Relação às Arquiteturas de Computador

Considerando-se a revisão apresentada sobre as arquiteturas de computador, verifica-se que o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções existe e pode ser considerador relevante em todas as

arquitetura citadas. Assim, para todas elas, CISC, RISC, Superescalar, Superpipeline e VLIW, uma boa solução para o problema da interdependência poderia contribuir para a obtenção de um melhor desempenho dos programas. Isso porque, em todas as arquiteturas, tanto a alocação de registradores, quanto o escalonamento de instruções precisam ser realizados em algum momento e de alguma maneira, seja pelo compilador exclusivamente, seja pelo compilador e pela máquina.

Considerando as características de cada uma das arquiteturas apresentadas, vemos que as arquiteturas superescalares, que utilizam iniciação fora de ordem, realizam o escalonamento das instruções dinamicamente, utilizando o próprio *hardware*, de modo que o escalonamento produzido pelo compilador é muito pouco relevante para essas arquiteturas. Mais ainda, como a tecnologia superescalar pode ser utilizada por qualquer outra arquitetura e, de fato, tem sido aplicada nas arquiteturas de quase todos os processadores modernos que exigem alto desempenho [Stallings, 2010], todos esses processadores também não necessitarão do escalonamento estático de instruções feito pelo compilador, desde que a iniciação de instruções fora de ordem seja realizada.

Com relação à tarefa de alocação de registradores, é razoável considerar que, sob o aspecto da otimização do código, ela será útil sempre que o número de registradores disponíveis para atribuição for reduzido, uma vez que, nos casos contrários, todos os pseudoregistradores poderão ser atribuídos a um registrador sem grandes dificuldades. Nesse sentido, é possível constatar que ela é relevante para todas as arquiteturas, pois todas possuem um número limitado de registradores disponíveis. Mesmo nas arquiteturas puramente RISC, que possuem um número bem elevado de registradores, como a arquitetura SPARC por exemplo, que pode ter até 520 registradores de propósito geral, esse número é limitado, em função da utilização de janelas de registradores.

Sendo assim, do ponto de vista teórico e em relação às arquiteturas de computador consideradas, poderíamos situar o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções da seguinte maneira: possivelmente, trata-se de um problema que será melhor observado nas arquiteturas que não são capazes de escalonar dinamicamente as instruções do programa, isto é, nas arquiteturas que, mesmo utilizando tecnologia superescalar, não iniciam instruções fora de ordem, uma vez que, nessas arquiteturas, o código gerado pelo compilador é exatamente igual ao código executado pela máquina, sem modificações. Nas arquiteturas capazes de iniciar instruções fora de ordem, o escalonamento produzido pelo compilador pode ser bastante alterado e, devido às técnicas de otimização utilizadas durante esse processo de alteração, e à alta velocidade em que ele é realizado, os efeitos de um algoritmo utilizado pelo compilador para tratar o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções podem ser

mascarados, ou mesmo modificados.

Todavia, é de se esperar que o adequado tratamento do problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções produza efeitos positivos no desempenho dos código gerados para quaisquer arquiteturas. Para aquelas que respeitam a ordem de instruções definida pelo compilador, o desempenho do código estará diretamente relacionado à qualidade do código produzido, considerando os resultados das tarefas de alocação e escalonamento. Para as arquiteturas capazes de iniciar instruções fora de ordem, as modificações que a máquina precisará fazer no código serão tão menores, quanto melhor for o código gerado pelo compilador em termos da utilização dos registradores e das possibilidades de paralelismo, e o desempenho final se beneficiará desse menor número de modificações.

2.4 Alocação de Registradores

O código intermediário gerado pelo *frontend* utiliza em suas instruções variáveis temporárias, *pseudoregistradores*, para armazenar os resultados parciais das suas operações.

Considerando que o acesso aos registradores da máquina é muito mais eficiente, tanto em termos de velocidade, quanto em termos de consumo energético, do que o acesso à memória [Aho et al., 2006; Luna et al., 2015], o código gerado pelo compilador será muito mais eficiente, se os valores das suas variáveis temporárias estiverem armazenados nos registradores.

A alocação de registradores é a tarefa responsável por esse armazenamento. Ela mapeia cada um dos pseudoregistradores do código intermediário para um registrador da máquina alvo e é composta por dois processos distintos: *alocação* e *atribuição* [Aho et al., 2006]. O processo de alocação é responsável por selecionar os pseudoregistradores do programa que serão mantidos nos registradores da máquina. O processo de atribuição, por sua vez, atribui cada um dos pseudoregistradores escolhidos a um determinado registrador. Nesta dissertação, o termo *alocação de registradores* será utilizado para se referir ao conjunto desses dois processos indistintamente.

A associação de um pseudoregistrador do código intermediário a um determinado registrador nem sempre é possível. Há casos em que a quantidade de registradores da máquina é menor do que a quantidade de pseudoregistradores do código, fazendo com que alguns desses pseudoregistradores precisem ser armazenados na memória do computador. Nesses casos, dizemos que houve derramamento (*spill*) de código para a memória e, para cada um desses derramamentos, instruções adicionais de escrita e leitura na memória, do tipo `load` e `store`, precisarão ser inseridas no programa objeto.

O resultado da tarefa de alocação de registradores pode ser considerado ótimo, quando o número de pseudoregistradores alocados é máximo e o número de registradores utilizados é mínimo. Além disso, idealmente, os pseudoregistradores devem permanecer alocados durante todo o seu *tempo de vida*, isto é, durante todo o tempo em que seus valores forem necessários ao programa.

2.4.1 Alocação de Registradores via Coloração de Grafos

A abordagem clássica para a solução do problema de alocação de registradores é considerada a técnica de coloração de grafos proposta em Chaitin et al. [1981]. Outros métodos baseados nessa abordagem também foram propostos, tais como Briggs [1992] e Appel [1998]. Nessa abordagem, para modelar as interferências existentes entre os tempos de vida das variáveis, o alocador de registradores utiliza um grafo não dirigido, denominado *grafo de interferência*. Os vértices desse grafo representam os pseudoregistradores do código. Dois vértices estarão conectados por uma aresta sempre que estiverem vivos em algum trecho de código simultaneamente. Assim, os pseudoregistradores representados por vértices adjacentes no grafo, vértices que incidem sobre uma mesma aresta, não podem ser mapeados para um mesmo registrador.

Dizemos que um grafo que pode ser colorido com K cores é *K-colorível*, e chamamos de *K-coloração* qualquer uma das possibilidades de colori-lo com K cores. Uma K-coloração de um grafo *é válida*, se quaisquer dois vértices adjacentes possuem cores distintas. Uma K-coloração *é ótima*, se ela utiliza o menor número de cores dentre todas as colorações válidas possíveis. Nesse caso, dizemos que K é o número cromático do grafo, ou seja, K é o menor número de cores com o qual podemos ter uma coloração válida do grafo. Assim, uma K-coloração ótima do grafo de interferência implica uma alocação ótima dos registradores e ela ocorrerá sempre que o número de registradores da máquina não for menor do que o número cromático do grafo de interferência.

O objetivo da abordagem clássica é colorir o grafo de interferência com um número de cores menor ou igual a K, sendo K o número de registradores da máquina. Como esse problema pode ser reduzido ao problema de coloração mínima de um grafo, e este último é um problema NP-completo [Karp, 1972; Garey & Johnson, 1979], podemos deduzir que o problema de alocação de registradores também é NP-completo. Na prática, diferentes heurísticas foram propostas e utilizadas, para produzir soluções próximas à solução ótima [Briggs et al., 1994, 1992, 1989; Chaitin, 1982; Appel, 1998].

Considerando que o número de registradores de uma máquina é geralmente menor do que o número de pseudoregistradores do código intermediário, frequentemente é necessário que, no processo de coloração do grafo, algumas variáveis sejam derramadas

para a memória, a fim de simplificar o grafo de interferência, tornando-o K-colorível. Por esse motivo, podemos também dizer que o alocador de registradores tem o objetivo de minimizar o custo total de derramamentos (*spills*) para a memória. Nesse sentido, vários métodos e heurísticas também foram criados [Silva, 2015; Quintão Pereira & Palsberg, 2008; Barany & Krall, 2013; Gao & Shi, 2005; Koseki et al., 2003; Govindarajan et al., 2003; Cooper & Taylor Simpson, 1998; Bergner et al., 1997; Bernstein et al., 1989].

2.4.1.1 Método de Chaitin

Chaitin et al. [1981] foi o primeiro a utilizar a técnica de coloração de grafos para realizar a tarefa de alocação de registradores. O método definido constrói o grafo de interferência $G = (V, E)$, sendo V o conjunto de *tempos de vida* e E o conjunto de interferências entre cada par de *tempos de vida* existentes em V. O método admite a existência de K cores para colorir o grafo, sendo K o número de registradores da máquina alvo. A partir daí, tenta encontrar uma *K-coloração* ótima para G. Todas as expressões temporárias, subexpressões comuns e variáveis associadas aos vértices de G que foram coloridos são mapeados para registradores da máquina. As variáveis associadas aos vértices que não puderam ser coloridos são derramadas para a memória.

No método de Chaitin et al. e também nos métodos de Briggs [1992] e Bernstein et al. [1989], uma vez feito o mapeamento, os pseudoregistradores permanecem nos registradores ou na memória durante todo o tempo de execução do bloco.

O método de Chaitin et al. é ilustrado na Figura 2.7 e pode ser dividido nas sete seguintes fases:

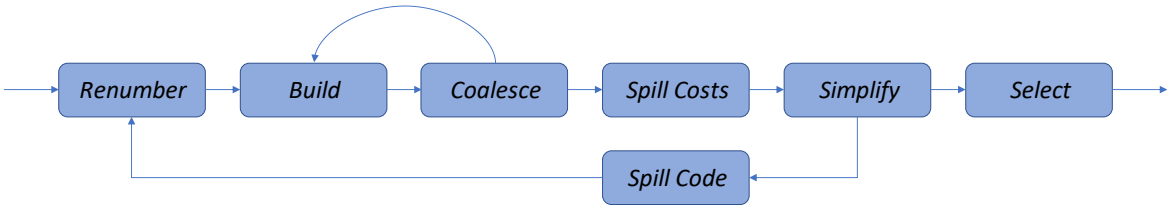


Figura 2.7. Fases do Método de Chaitin.

- *Renumber*: encontra e identifica os *tempos de vida* dos pseudoregistradores.
- *Build*: constrói o grafo de interferência.
- *Coalesce*: tenta remover instruções de cópia, utilizando um único registrador para os diferentes *tempos de vida* presentes nessas instruções.

- *Spill Costs*: determina para cada pseudoregistrador o seu custo de derramamento, que é uma estimativa do número de instruções `load` e `store` necessárias para alocar o pseudoregistrador na memória.
- *Simplify*: elimina do grafo de interferência os vértices com grau menor do que K. Cada vértice eliminado é colocado em uma pilha de seleção S. Se nenhum vértice puder ser eliminado, o vértice com menor prioridade será retirado do grafo e colocado na lista de derramamento SL. A prioridade de um vértice é calculada dividindo-se o seu custo de derramamento pelo seu grau, que é o total de vértices adjacentes que ele possui. O algoritmo prossegue até que todos os vértices tenham sido retirados do grafo. Caso nenhum vértice tenha sido colocado na lista de derramamento, o algoritmo passa para a fase *Select*. Caso contrário, a fase *Spill Code* é executada.
- *Select*: desempilha os vértices que estão na pilha S e atribui a cada um deles a primeira cor disponível e não utilizada por seus vizinhos.
- *Spill Code*: o algoritmo reescreve o código intermediário, acrescentando as instruções `load` e `store` para cada um dos pseudoregistradores presentes na lista SL, e volta para a fase *Renumber*.

2.4.1.2 Método de Briggs

Briggs [1992] estendeu o método de Chaitin et al. introduzindo técnicas que diminuíram o número de derramamentos de pseudoregistradores para a memória [Briggs et al., 1989, 1992, 1994]. Dentre essas técnicas, destacamos a que ficou conhecida como *Optimistic Coloring* [Briggs et al., 1989], pela qual a execução da fase *Spill Code* do método é adiada. A Figura 2.8 ilustra a sequência de fases do método Briggs.

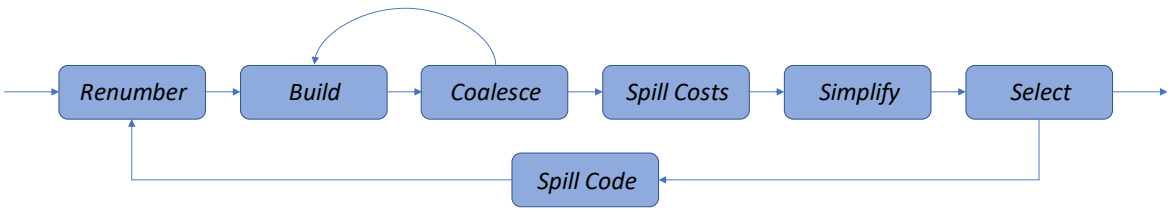


Figura 2.8. Fases do Método de Briggs.

Com *Optimist Coloring*, após a fase *Simplify*, mesmo que haja vértices na lista de derramamento SL, a fase *Select* é acionada e, após a atribuição de cores para os vértices que estão na pilha de seleção S, verifica-se a disponibilidade de cores para cada um dos vértices que eventualmente estão na lista de derramamento SL. "Com sorte",

alguma cor estará disponível e o vértice poderá ser retirado da lista. Caso todos os vértices tenham sido retirados da lista, o método é finalizado. Caso contrário, a fase *Spill Code* é acionada.

2.4.2 Outros Métodos e Técnicas de Minimização de *Spill*

Além dos métodos clássicos para a solução do problema de alocação de registradores, outros métodos foram propostos. Quintão Pereira & Palsberg [2008] propuseram um método que modela o conjunto de registradores como o tabuleiro de um quebra-cabeça, e os pseudoregistradores como as peças do quebra-cabeça a ser montado. Para determinadas arquiteturas, esse método solucionou o problema de alocação em tempo polinomial. Ambrosio et al. [2004] apresentaram um método baseado em Crescimento de Domínios Ativos e Combinação de Registradores. Apesar de mais caro do que os métodos clássicos, esse método se mostrou mais eficiente, gerando menos derramamentos para a memória nos casos em que existem muitos laços e muitas variáveis globais vivas durante todo o programa.

Desde a proposta de Chaitin et al. [1981], várias tentativas de melhorar a solução clássica têm sido propostas, cada uma delas definindo e utilizando diferentes heurísticas e técnicas para a redução do número de derramamentos (Chaitin [1982]; Bernstein et al. [1989]; Briggs et al. [1992, 1994]; Bergner et al. [1997]; Cooper & Taylor Simpson [1998]; Govindarajan et al. [2003]; Koseki et al. [2003]; Chow & Hennessy [2004]; Gao & Shi [2005]; Barany & Krall [2013]; dentre outros). Mais recentemente, Silva [2015] apresentou uma nova técnica denominada *Color Flipping*, que busca minimizar os custos de derramamento recolorindo os vértices do grafo de interferência. Ela utiliza o método de Briggs [1992] para selecionar as variáveis que serão derramadas para a memória, contudo, antes que o derramamento seja realizado, o algoritmo *color flipping* é invocado, a fim de testar novas possibilidades de cores para os vértices do grafo de interferência, visando a redução do número de derramamentos.

2.5 Escalonamento de Instruções

As arquiteturas dos processadores de alto desempenho permitem que eles trabalhem com paralelismo em nível de instrução, a fim de tornar a execução dos programas mais rápida. Essa capacidade pode ser alcançada com a utilização de vários processadores, com a implementação de várias unidades funcionais no processador, ou com a implementação da técnica de *pipelining* [Aho et al., 2006]. Entretanto, para que essa capacidade seja utilizada, é necessário que as instruções que podem ser paralelizadas

sejam devidamente identificadas e escalonadas. Esse é o objetivo do escalonador de instruções [Aho et al., 2006].

Escalonamento de instruções é o processo de identificar e mover as instruções do código, alterando sua sequência original de execução, de tal modo que elas possam ser paralelizadas, minimizando o tempo de execução do programa. Segundo M. Bigonha [1994], para que essa movimentação de instruções seja feita, dois pontos fundamentais devem ser considerados: a semântica do programa deve ser preservada e uma melhor utilização da máquina deve ser alcançada.

A estrutura principal utilizada pelo escalonador de instruções é o *grafo de escalonamento*, que é construído de acordo com os seguintes critérios: a) cada um dos vértices do grafo corresponde a uma instrução do código intermediário; b) existe uma aresta dirigida *(u,v)* de u para v, se a instrução do vértice u deve ser executada antes da instrução do vértice v. Isso ocorre somente nos seguintes casos: (i) existe uma dependência de dados de u para v; (ii) existe uma dependência de controle de u para v; (iii) existe uma restrição de máquina que força a execução de u antes da execução de v [M. Bigonha, 1994].

Um escalonador de instruções que rearranja o código dentro de cada bloco básico é chamado escalonador local, enquanto aquele que movimenta instruções entre blocos básicos, levando em consideração o efeito global dessa movimentação, é conhecido como escalonador global. No caso de um escalonador global, o grafo de escalonamento é o grafo de fluxo de controle, no qual cada vértice representa um bloco básico do programa.

Em qualquer situação, seja no escalonamento local ou global, um escalonamento válido deve preservar a ordem de execução definida pelas arestas do grafo de escalonamento. Assim, a atribuição de precedências às arestas do grafo de escalonamento pode restringir o potencial de paralelismo do escalonador. Tais precedências podem ser necessárias em virtude de dependências de dados, dependências de controle, ou restrições de máquina.

Na literatura, o escalonamento de instruções tem sido realizado, principalmente, de duas formas distintas: *loop scheduling*, ou escalonamento de laço, quando se busca escalonar as instruções internas a cada um dos laços do programa; e *basic block scheduling*, ou escalonamento de blocos básicos, quando se deseja realizar o escalonamento local. O algoritmo *Modulo Scheduling*, proposto por Rau [1994], tem sido o mais utilizado para a realização do escalonamento de laço. Huff [1993] e Zalamea et al. [2001] propuseram várias modificações desse algoritmo e um estudo comparativo entre várias técnicas de *Modulo Scheduling* pode ser encontrado em Codina et al. [2002]. Para o escalonamento local, de acordo com Kim & Lee [2010], a abordagem mais utilizada tem sido a *lista de escalonamento* [Fisher, 1981; Fisher et al., 1984; Gibbons & Muchnick,

1986; Goodman & Hsu, 1988], na qual o escalonador mantém uma lista de instruções aptas a serem escalonadas sem provocar atrasos. A cada iteração do escalonador, uma instrução da lista é selecionada, por meio de uma heurística de seleção, retirada da lista e executada.

O problema do escalonamento de instruções tem sido considerado NP-difícil [Kri & Feeley, 2004; Leupers, 2000], mas pode se tornar NP-completo na presença de certas características e restrições [van Beek & Wilken, 2001]. Seja como for, em ambos os casos, ele não possui uma solução ótima que possa ser encontrada em tempo polinomial.

2.6 O Problema da Interdependência entre AR e EI

As tarefas de alocação de registradores e escalonamento de instruções, em última análise, objetivam a produção de um programa com melhor desempenho. Cada uma delas, a seu próprio modo, busca minimizar a duração total do escalonamento gerado, isto é, a duração total da execução do código. A primeira, a tarefa de alocação, procura utilizar adequadamente os registradores da máquina e evitar derramamentos de variáveis para a memória. A segunda, a tarefa de escalonamento, visa sequenciar as instruções do programa de tal modo a aproveitar ao máximo a capacidade da máquina de paralelizar instruções.

Contudo, essas duas tarefas estão envolvidas em um problema de priorização, que pode ser descrito da seguinte maneira: se a alocação de registradores for a primeira tarefa a ser realizada, variáveis de instruções independentes, que poderiam ser executadas em paralelo, podem ser atribuídas a um mesmo registrador da máquina, limitando as possibilidades de escalonamento e prejudicando o desempenho do programa; de outro modo, se o escalonamento de instruções for feito antes da alocação, o tempo de permanência das variáveis nos registradores pode ser muito longo e, em função dessa excessiva ocupação, o número de acessos à memória pode aumentar, prejudicando dessa maneira o desempenho do código gerado.

Assim, além das tarefas de alocação e escalonamento serem por si só problemas de difícil solução, a interferência que elas exercem entre si também representa um árduo problema, para o qual uma solução ótima e geral ainda não foi encontrada.

Alguns dos métodos que buscam solucionar esse problema realizam o escalonamento de instruções antes da alocação de registradores, outros fazem exatamente o contrário e escalonam as instruções do código somente após a alocação dos registradores ter sido feita. Na literatura, desde Hennessy & Gross [1983], essas duas formas de abordagem são conhecidas como *prepass* e *postpass* respectivamente.

Conforme Goodman & Hsu [1988] e de acordo com a descrição apresentada para o problema da interdependência, as abordagens *prepass* dão ao compilador a vantagem de explorar todo o paralelismo de instruções existente no código, ao passo que as abordagens *postpass* possibilitam a efetiva minimização do número de derramamentos de variáveis para a memória.

As desvantagens de cada uma dessas abordagens são a própria essência do problema da interdependência. Assim, tais desvantagens já foram indicadas na descrição feita sobre esse problema.

Nos métodos *prepass*, o número de acessos à memória pode ser elevado e essa desvantagem ocorre quando, em função do deslocamento de instruções feito pelo escalonador, um registrador da máquina é forçado a armazenar um único valor por um longo trecho do código, fazendo com que outros valores, que poderiam ser armazenados nele, se ele estivesse disponível, precisem ser enviados para a memória.

Nos métodos *postpass*, as possibilidades de escalonamento podem ser limitadas e tal desvantagem ocorre em função das dependências de dados que podem surgir entre as instruções do código após a alocação dos registradores. Tais dependências não existiam no código original e não são *dependências verdadeiras*, do tipo *leitura-após-escrita*. Elas não são causadas porque os dados estão sendo passados de uma instrução para outra, e sim porque o mesmo registrador está sendo utilizado em mais de uma instrução [Padua & Wolfe, 1986]. Elas poderiam ser removidas do código, caso diferentes registradores fossem utilizados nas instruções, e, por isso, são consideradas *falsas dependências*.

Segundo Aho et al. [2006], essas falsas dependências podem ser de dois tipos: *dependências escrita-após-leitura* ou *dependências de saída*. As do primeiro tipo, também conhecidas como WAR (*Write-After-Read*), ou *antidependências*, ocorrem quando uma instrução que utiliza o valor de um registrador é seguida por outra instrução que redefine o valor desse mesmo registrador. Nesse caso, a limitação no escalonamento acontece devido à impossibilidade de antecipação da instrução de escrita, pois, caso contrário, o valor disponível na instrução de leitura estaria incorreto. As dependências do segundo tipo, dependências de saída, surgem quando duas instruções de escrita compartilham o mesmo registrador. Nessa situação, o escalonador fica impossibilitado de alterar a ordem de chamada das instruções, a fim de não alterar o valor final que deve existir no registrador.

2.6.1 Exemplo

Considere que a Figura 2.9 (a) apresenta um código fonte que deve ser compilado. A Figura 2.9 (b) apresenta o código intermediário obtido pela tradução de cada uma das

instruções do fonte para um código de três endereços. Finalmente, a Figura 2.9 (c) apresenta um código intermediário semanticamente equivalente ao anterior, mas que já utiliza instruções de máquina. Uma tradução inicial para a linguagem de máquina foi realizada entre (b) e (c), contudo, o código em (c) não pode ser considerado o programa objeto final, pois ainda utiliza pseudoregistradores (*PR*) em suas instruções. Tal código foi gerado pelas etapas iniciais do gerador de código e está sendo entregue às tarefas de alocação de registradores e escalonamento de instruções neste momento[7]. A partir desse código, que chamaremos apenas de *código intermediário* deste ponto em diante deste exemplo, as abordagens *prepass* e *postpass* serão utilizadas para a geração do programa objeto.

```
A = a + b
B = c - d
C = e + f
D = 2 * C
E = A * B * C
F = E - D
```

(a)

```
t1 := a + b
t2 := c - d
t3 := e + f
t4 := 2 * t3
t5 := t1 * t2
t6 := t5 * t3
t7 := t6 - t4
```

(b)

```
1:  LOAD PR1, a
2:  LOAD PR2, b
3:  ADD  PR3, PR1, PR2     # t1
4:  LOAD PR4, c
5:  LOAD PR5, d
6:  SUB  PR6, PR4, PR5     # t2
7:  LOAD PR7, e
8:  LOAD PR8, f
9:  ADD  PR9, PR7, PR8     # t3
10: MUL  PR10, 2, PR9      # t4
11: MUL  PR11, PR3, PR6    # t5
12: MUL  PR12, PR11, PR9   # t6
13: SUB  PR13, PR12, PR10  # t7
14: STOR PR13, F           # F
```

(c)

Figura 2.9. Códigos para ilustrar o problema da interdependência: (a) Código fonte, (b) Código intermediário com instruções em um código de três endereços e (c) Código intermediário com instruções de máquina.

Abordagem *Prepass* Nessa abordagem, a primeira tarefa a ser realizada é o escalonamento de instruções. As Figuras 2.10 (a), (b) e (c) apresentam, respectivamente, o código intermediário, o código gerado pelo escalonador de instruções e o código gerado pelo alocador de registradores, que é o programa objeto final.

A Figura 2.11 apresenta o grafo de escalonamento relacionado ao código intermediário. Nesse grafo, a posição de cada instrução no sequenciamento feito está indicada no ícone amarelo posicionado no canto superior direito do vértice que representa a instrução. O escalonamento foi feito com o mesmo algoritmo utilizado em Goodman &

[7]Na maioria dos compiladores reais, como no caso do LLVM, a tradução para a linguagem de máquina é uma das últimas tarefas realizadas pelo compilador. Ela é feita somente após as tarefas de alocação de registradores e escalonamento de instruções terem sido realizadas. Todavia, a antecipação proposta neste exemplo não o invalida e foi utilizada para torná-lo mais didático e inteligível.

Hsu [1988]. Trata-se do algoritmo de lista de escalonamento descrito na Seção 2.5, que, basicamente, realiza no grafo uma busca em largura a partir do seu último vértice.

A partir do código gerado pelo escalonador, Figura 2.10 (b), a alocação dos registradores é realizada. A Figura 2.12 (a) apresenta a tabela de variáveis vivas ao longo desse código, e a Figura 2.12 (b) mostra o grafo de interferência construído a partir das informações dessa tabela. Os números dos registradores associados aos pseudoregistradores estão indicados nos ícones amarelos presentes no canto superior direito de cada vértice do grafo. O algoritmo de Chaitin foi utilizado para colorir o grafo e assim determinar os registradores a serem utilizados.

Como resultado final, o programa objeto gerado com a abordagem *prepass*, Figura 2.10 (c), aproveita todas as possibilidades de paralelismo do código intermediário, mas utiliza seis registradores.

```
1:  LOAD PR1, a
2:  LOAD PR2, b
3:  ADD  PR3, PR1, PR2     # t1
4:  LOAD PR4, c
5:  LOAD PR5, d
6:  SUB  PR6, PR4, PR5     # t2
7:  LOAD PR7, e
8:  LOAD PR8, f
9:  ADD  PR9, PR7, PR8     # t3
10: MUL  PR10, 2, PR9      # t4
11: MUL  PR11, PR3, PR6    # t5
12: MUL  PR12, PR11, PR9   # t6
13: SUB  PR13, PR12, PR10  # t7
14: STOR PR13, F           # F
```

(a)

```
1:  LOAD PR1, a
2:  LOAD PR2, b
4:  LOAD PR4, c
5:  LOAD PR5, d
7:  LOAD PR7, e
8:  LOAD PR8, f
3:  ADD  PR3, PR1, PR2     # t1
6:  SUB  PR6, PR4, PR5     # t2
9:  ADD  PR9, PR7, PR8     # t3
11: MUL  PR11, PR3, PR6    # t5
10: MUL  PR10, 2, PR9      # t4
12: MUL  PR12, PR11, PR9   # t6
13: SUB  PR13, PR12, PR10  # t7
14: STOR PR13, F           # F
```

(b)

```
1:  LOAD R1, a
2:  LOAD R2, b
4:  LOAD R3, c
5:  LOAD R4, d
7:  LOAD R5, e
8:  LOAD R6, f
3:  ADD  R2, R1, R2    # t1
6:  SUB  R4, R3, R4    # t2
9:  ADD  R5, R5, R6    # t3
11: MUL  R2, R2, R4    # t5
10: MUL  R1, 2, R5     # t4
12: MUL  R2, R2, R5    # t6
13: SUB  R1, R2, R1    # t7
14: STOR R1, F         # F
```

(c)

Figura 2.10. Resultado da abordagem *prepass*: (a) Código intermediário, (b) Código após o escalonamento das instruções e (c) Código objeto final, após a alocação dos registradores.

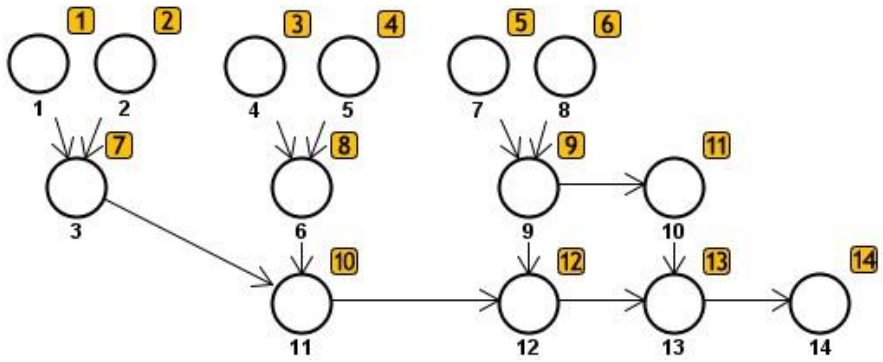


Figura 2.11. Abordagem *prespass*: grafo de escalonamento.

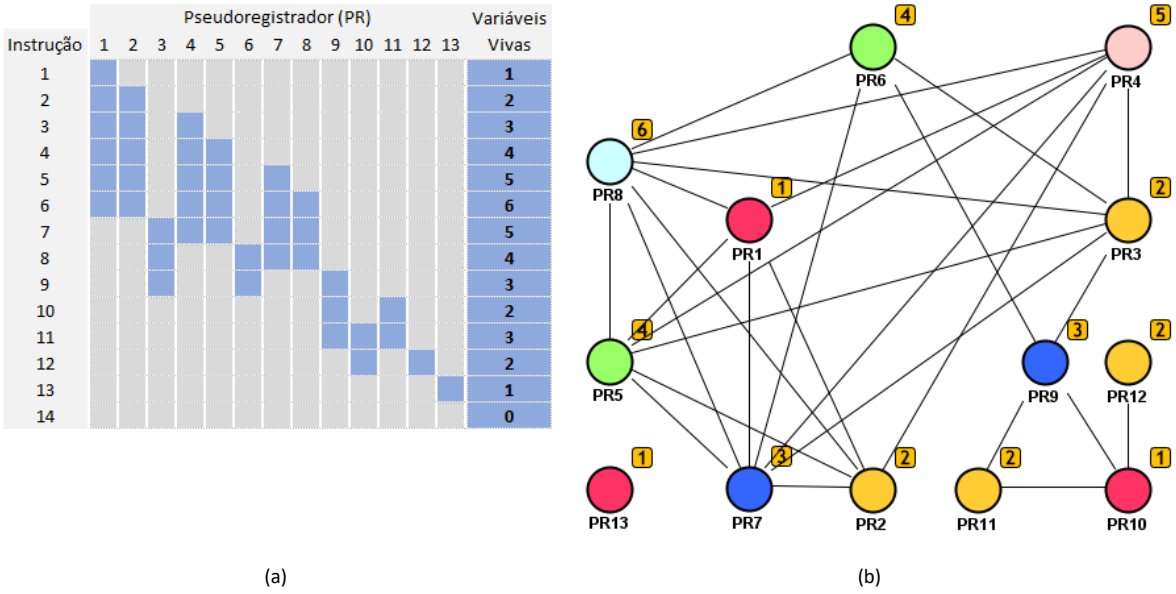


Figura 2.12. Abordagem *prespass*: (a) variáveis vivas; (b) grafo de interferência.

Abordagem *Postpass* Nessa abordagem, a alocação de registradores é a primeira tarefa a ser executada. As Figuras 2.13 (a), (b) e (c) apresentam, respectivamente, o código intermediário, o código gerado pelo alocador de registradores e o código gerado pelo escalonador de instruções, que é o programa objeto final.

A Figura 2.14 (a) apresenta a tabela de variáveis vivas ao longo do código intermediário, e a Figura 2.14 (b) mostra o grafo de interferência construído a partir das informações dessa tabela. Assim como foi feito na abordagem anterior, os números dos registradores associados aos pseudoregistradores estão indicados nos ícones próximos de cada um dos vértices do grafo.

A partir do código gerado pelo alocador de registradores, Figura 2.13 (b), o escalonamento das instruções é realizado. A Figura 2.15 apresenta o grafo de escalonamento relacionado a esse código. Nele, a posição de cada instrução no sequenciamento está indicada no canto superior direito de cada vértice.

Os algoritmos utilizados para colorir o grafo de interferência e para escalonar as instruções foram os mesmos utilizados na abordagem *prepass*. Com relação ao escalonamento, para a compreensão do resultado encontrado, é importante ressaltar que as arestas relativas às dependências de saída, arestas pontilhadas no grafo e marcadas com as iniciais *DS*, não são levadas em consideração pelo algoritmo, enquanto as arestas de antidependência, marcadas com as iniciais *WAR*, são consideradas por ele.

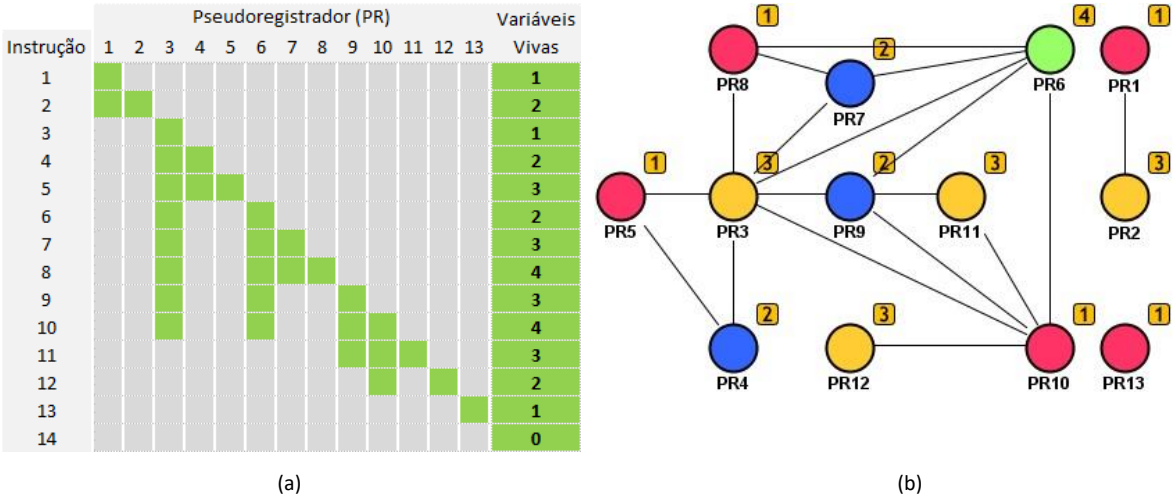
Como resultado final, o programa objeto gerado com a abordagem *postpass*, Figura 2.13 (c), utiliza apenas quatro registradores, mas não aproveita todas as possibi-

lidades de paralelismo existentes no código intermediário.

```
1:  LOAD PR1, a                    1:  LOAD R1, a                    1:  LOAD R1, a
2:  LOAD PR2, b                    2:  LOAD R3, b                    2:  LOAD R3, b
3:  ADD  PR3, PR1, PR2     # t1    3:  ADD  R3, R1, R3    # t1       3:  ADD  R3, R1, R3    # t1
4:  LOAD PR4, c                    4:  LOAD R2, c                    4:  LOAD R2, c
5:  LOAD PR5, d                    5:  LOAD R1, d                    5:  LOAD R1, d
6:  SUB  PR6, PR4, PR5     # t2    6:  SUB  R4, R2, R1    # t2       6:  SUB  R4, R2, R1    # t2
7:  LOAD PR7, e                    7:  LOAD R2, e                    7:  LOAD R2, e
8:  LOAD PR8, f                    8:  LOAD R1, f                    8:  LOAD R1, f
9:  ADD  PR9, PR7, PR8     # t3    9:  ADD  R2, R2, R1    # t3       9:  ADD  R2, R2, R1    # t3
10: MUL  PR10, 2, PR9      # t4    10: MUL  R1, 2, R2     # t4       11: MUL  R3, R3, R4    # t5
11: MUL  PR11, PR3, PR6    # t5    11: MUL  R3, R3, R4    # t5       10: MUL  R1, 2, R2     # t4
12: MUL  PR12, PR11, PR9   # t6    12: MUL  R3, R3, R2    # t6       12: MUL  R3, R3, R2    # t6
13: SUB  PR13, PR12, PR10  # t7    13: SUB  R1, R3, R1    # t7       13: SUB  R1, R3, R1    # t7
14: STOR PR13, F           # F     14: STOR R1, F         # F        14: STOR R1, F         # F
```

(a) (b) (c)

Figura 2.13. Resultado da abordagem *postpass*: (a) Código intermediário, (b) Código após a alocação dos registradores e (c) Código objeto final, após o escalonamento das instruções.



(a) (b)

Figura 2.14. Abordagem *postpass*: (a) variáveis vivas; (b) grafo de interferência.

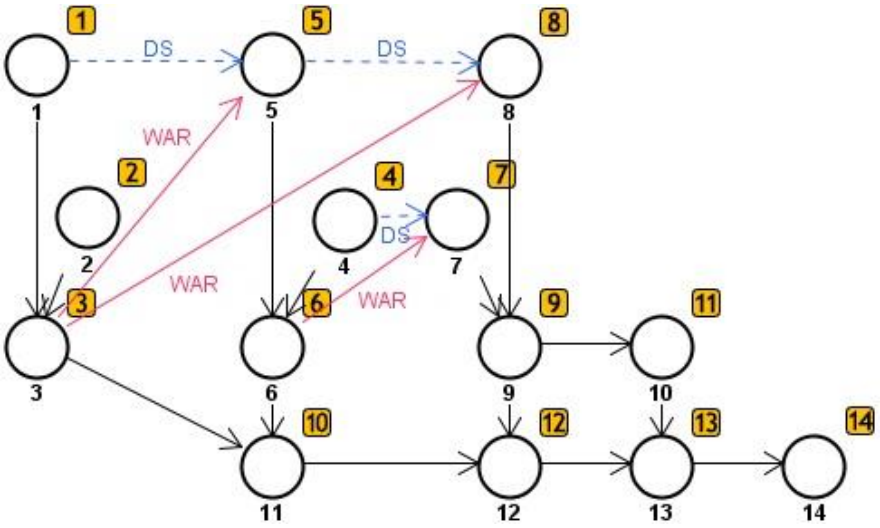


Figura 2.15. Abordagem *postpass*: grafo de escalonamento.

Análise dos Resultados do Exemplo

Comparando-se os resultados das duas abordagens, observa-se como o escalonamento *prepass* é bem mais eficiente do que o escalonamento *postpass*. Com efeito, este último quase não transformou o código intermediário. Como mostram as Figuras 2.13 (b) e (c), a sequência de instruções do escalonamento *postpass* difere da que já existia no código intermediário apenas pela posição das instruções 10 e 11.

Por outro lado, o escalonamento *prepass* aumenta o tempo de vida dos pseudoregistradores, fazendo com que a interferência que eles exercem entre si também aumente. Comparando-se as tabelas de variáveis vivas de cada abordagem, Figuras 2.12 (a) e 2.14 (a), nota-se que a maior parte dos pseudoregistradores possui maior tempo de vida na abordagem *prepass*: sete dentre os treze existentes, cerca de 54% deles, possuem maior tempo de vida nessa abordagem.

Isso explica o resultado inferior da tarefa de alocação de registradores na abordagem *prepass*. Nessa abordagem, seis registradores foram utilizados, ao passo que apenas quatro registradores foram necessários na abordagem *postpass*. Em termos relativos, trata-se de uma expressiva diferença de 50%. Se a máquina não possuir mais do que quatro registradores, algumas das variáveis precisarão ser derramadas para a memória na abordagem *prepass*.

Finalmente, é importante destacar que o bom desempenho da tarefa de alocação de registradores na abordagem *postpass* faz surgir entre as instruções do código uma série de falsas dependências, devido a elevada reutilização dos registradores. Essas falsas dependências estão representadas no grafo da Figura 2.15 pelas arestas *WAR* e

DS e são a causa do resultado inferior da tarefa de escalonamento nessa abordagem.

2.7 Considerações Finais

Neste capítulo, apresentamos os principais conceitos e as principais teorias relacionadas ao restante desta dissertação.

Na Seção 2.1, foram descritos os conceitos de bloco básico, código de três endereços e diagrama de fluxo de controle, que são básicos para a teoria de compiladores. As Seções 2.2 e 2.3 introduziram as teorias de otimização e geração de código respectivamente, e as Seções 2.4 e 2.5 forneceram explicações mais aprofundadas sobre as tarefas de alocação de registradores e escalonamento de instruções. Por fim, a Seção 2.6 apresentou detalhadamente o problema da interdependência entre essas duas tarefas, que é o tema da pesquisa realizada.

O próximo capítulo apresenta a revisão sistemática da literatura que foi realizada sobre o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, investigando as abordagens e os métodos utilizados para lidar com ele.

Capítulo 3

Revisão Sistemática da Literatura

Nos últimos anos, várias pesquisas foram realizadas sobre alocação de registradores e escalonamento de instruções, bem como sobre a interdependência entre essas duas tarefas. A fim de identificar e resumir tais pesquisas e visando um adequado posicionamento deste trabalho no contexto atual, realizamos uma *Revisão Sistemática da Literatura* (RSL) sobre o tema desta dissertação, ou seja, sobre a interdependência entre as tarefas de alocação de registradores e escalonamento de instruções.

Neste capítulo, apresentamos o processo adotado para a realização dessa revisão sistemática da literatura, bem como os resultados obtidos a partir dela. A principal bibliografia utilizada para a definição das fases e atividades dessa revisão sistemática foram os relatórios técnicos de Kitchenham & Charters [Kitchenham, 2004; Kitchenham & Charters, 2007]. Contudo, outros textos relacionados a metodologias de pesquisa, revisões sistemáticas e outros tipos de revisão também foram consultados [Fink, 2010; Petersen et al., 2008; Mian et al., 2005; Jung, 2004].

Segundo Kitchenham [2004], "*uma revisão sistemática de literatura é uma forma de identificar, avaliar e interpretar todas as pesquisas disponíveis que são relevantes a uma particular questão de pesquisa, área, ou fenômeno de interesse*". Os estudos identificados pela RSL são classificados como estudos primários, enquanto a própria RSL é uma forma de estudo secundário, uma vez que revisa todos os estudos primários com o objetivo de reunir e sintetizar as evidências relacionadas ao tema pesquisado.

De acordo com Kitchenham & Charters [2007], dentre as principais motivações para a realização de uma revisão sistemática, destacam-se as seguintes: (i) resumir as evidências existentes sobre um determinado tema de pesquisa ou tecnologia; (ii) identificar possíveis lacunas nas pesquisas atuais, a fim de sugerir áreas a serem investigadas; (iii) posicionar de forma apropriada novas atividades de pesquisa; (iv) examinar a correlação entre as evidências empíricas e as hipóteses teóricas; (v) suportar a criação de

novas hipóteses teóricas sobre um determinado tema.

Uma revisão sistemática difere de uma revisão simples e convencional, produzida por especialistas ou não, na medida em que utiliza uma abordagem científica, replicável e transparente, com o objetivo de eliminar interpretações tendenciosas sobre o tema pesquisado. Kitchenham & Charters enumeram algumas características que tornam uma revisão sistemática diferente de uma revisão tradicional:

(i) utiliza um protocolo de revisão, que nada mais é do que um plano que descreve como a revisão sistemática será conduzida, especificando as questões de pesquisa que serão respondidas e os métodos que serão utilizados;

(ii) define uma estratégia de busca com o objetivo de detectar o maior número possível de estudos primários relevantes;

(iii) documenta sua estratégia de busca, de modo que ela possa ser verificada e reproduzida por outros pesquisadores;

(iv) avalia cada um dos estudos primários encontrados de acordo com critérios de inclusão e exclusão previamente definidos;

(v) especifica as informações que deverão ser extraídas de cada estudo primário, incluindo os critérios qualitativos pelos quais tais estudos serão avaliados;

(vi) é um pré-requisito para meta-análises[1] quantitativas.

A revisão sistemática descrita neste trabalho de mestrado foi realizada em quatro fases principais: *planejamento*, *execução*, *análise* e *apresentação*.

Na fase de *planejamento* todos os elementos necessários à execução da revisão sistemática foram definidos, ou seja, definiu-se as questões de pesquisa a serem respondidas, as bases de dados a serem pesquisadas e os métodos de busca e seleção dos estudos primários, para a formação do conjunto de estudos para posterior avaliação.

Na fase de *execução*, realizou-se o processo de busca e seleção dos estudos primários de acordo com o planejamento previamente feito. A busca foi realizada nas bases de dados selecionadas, utilizando-se o método de busca definido. Os estudos primários obtidos foram então submetidos ao método de seleção definido, criando-se assim o conjunto de estudos para avaliação.

Na fase de *análise*, os estudos primários selecionados para avaliação foram lidos e resumidos, visando a extração dos seus principais dados. Os dados extraídos foram então analisados, organizados e sintetizados, possibilitando a determinação das repostas para as questões de pesquisa formuladas na fase de planejamento.

[1]Procedimentos estatísticos para a síntese dos resultados de diferentes estudos.

Finalmente, na fase de *apresentação*, um relatório com os resultados obtidos na revisão sistemática foi elaborado. Tal relatório, escrito em forma de artigo técnico, foi submetido a um congresso científico ligado ao tema pesquisado, visando a disseminação dos resultados dentro da comunidade de interesse.

Nas próximas seções deste capítulo, descrevemos as atividades e os resultados de cada uma das fases da RSL realizada. Após a descrição dessas fases, a Seção 3.5 apresenta as ameaças à validade dessa revisão sistemática e a Seção 3.6 apresenta nossas considerações finais sobre a RSL feita e seus resultados.

3.1 Planejamento

Esta seção apresenta a fase de planejamento da revisão sistemática da literatura. Nessa fase, todos os elementos do protocolo de revisão são definidos, ou seja, define-se aqui o plano que descreve como a revisão sistemática deverá ser conduzida. Os componentes principais desse plano são os seguintes: (a) as questões de pesquisa que deverão ser respondidas a partir da revisão sistemática realizada; (b) as bases de dados que serão consultadas para a busca dos estudos primários; (c) o método de busca que será utilizado em cada uma dessas bases; (d) o método de seleção dos estudos primários obtidos no processo de busca, incluindo os critérios de inclusão e exclusão que serão utilizados; (e) especificação das informações que deverão ser extraídas dos estudos primários.

3.1.1 Questões de Pesquisa

Com a revisão sistemática realizada, as três *Questões de Pesquisa* (QP) que se seguem deverão ser respondidas:

QP1: Quais são as principais abordagens propostas na literatura para lidar com o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções?

QP2: A consideração das tarefas de alocação de registradores e escalonamento de instruções *de forma conjunta* realmente melhora a otimização do código?

QP3: Para quais arquiteturas as tarefas de alocação de registradores e escalonamento de instruções têm sido realizadas *conjuntamente*?

3.1.2 Bases de Dados

As bases de dados escolhidas para a busca dos estudos primários estão listadas na Tabela 3.1. Elas foram escolhidas por serem bases de dados eletrônicas, bibliotecas virtuais acessíveis pela *internet*, com um grande acervo de trabalhos completos, publicados como artigos ou relatórios técnicos, em congressos e revistas relevantes na área de interesse dessa revisão sistemática.

Base de Dados	**Endereço Eletrônico**
ACM Digital Library	`http://dl.acm.org/`
IEEE	`http://ieeexplore.ieee.org/`
Science Direct	`http://www.sciencedirect.com/`
Scopus	`http://scopus.com/`
Springer	`http://link.springer.com/`
Web of Science	`http://webofknowledge.com/`

Tabela 3.1. Bases de dados eletrônicas selecionadas.

3.1.3 Método de Busca

Definiu-se que a busca dos estudos primários, em cada uma das bases de dados selecionadas, seria feita utilizando-se o motor de busca disponível na interface *web* de cada uma dessas bases.

Em função disso, um *termo de busca* foi elaborado, considerando as palavras-chave[2] relacionadas ao tema da revisão sistemática e também algumas pesquisas preliminares de teste, realizadas em cada uma das bases eletrônicas escolhidas. O *termo de busca* elaborado foi único para todas as bases, visando a padronização do processo de busca.

As palavras-chave para a criação do *termo de busca* foram escolhidas considerando a estrita relação com o tema da revisão sistemática. Nesta RSL, optamos por considerar apenas estudos primários publicados no idioma inglês. Assim sendo, para cada uma das palavras-chave selecionadas, um conjunto de palavras relacionadas no idioma inglês também foi escolhido. Todas essas palavras estão apresentadas na Tabela 3.2.

Definidas as palavras-chave e as palavras relacionadas, o *termo de busca* para a pesquisa foi criado utilizando-se os operadores lógicos `OR` e `AND` conforme os seguintes passos: 1) as *palavras relacionadas* a cada uma das três primeiras palavras-chave da Tabela 3.2 foram conectadas entre si com o operador lógico `OR`, de tal modo que três

[2]Neste contexto, como mostra a Tabela 3.2, palavras-chave são palavras simples propriamente ditas, ou expressões formadas por mais de uma palavra.

Palavras-chave	Palavras Relacionadas
Interdependência	*interdependence*, *interconnection*, *interrelationship*, *linkage*, *association*, *connection*, *correlation*, *relationship*, *combining*
Alocação de registrador	*register allocation*, *variable spilling*, *register packing*, *register pressure*, *register tiling*
Escalonamento de instrução	*instruction scheduling*, *scheduling*
Alocação de registradores e escalonamento de instruções	*register allocation and instruction scheduling*

Tabela 3.2. Palavras para o *termo de busca* da revisão sistemática.

grupos de palavras conectadas foram formados; 2) os três grupos de palavras criados no Passo 1 foram conectados entre si com o operador lógico `AND`, formando-se assim uma única estrutura; 3) a estrutura única criada no Passo 2 foi conectada com o operador lógico `OR` à *palavra relacionada* associada à última palavra-chave da Tabela 3.2. Assim, o seguinte *termo de busca* foi definido para a revisão sistemática:

((interdependence OR interconnection OR interrelationship OR linkage OR association OR connection OR correlation OR relationship OR combining) AND ("register allocation" OR "variable spilling" OR "register packing" OR "register pressure" OR "register tiling") AND ("instruction scheduling" OR scheduling)) OR ("register allocation and instruction scheduling")

Tabela 3.3. *Termo de busca* definido para a revisão sistemática.

3.1.4 Método de Seleção

O método definido para a seleção dos estudos primários foi baseado em quatro etapas e considerou critérios de inclusão e exclusão estipulados para a avaliação dos estudos primários.

O objetivo dos critérios de inclusão e exclusão é permitir a avaliação dos estudos primários obtidos no processo de busca. Para Mian et al. [2005], pesquisas executadas em motores de busca da *web* podem encontrar um grande número de trabalhos que não respondem às questões de pesquisa. Isso pode ocorrer porque uma palavra-chave pode ter significados diferentes, ou pode ser usada em estudos que não lidam com o tema de pesquisa explorado. Os critérios de inclusão e exclusão possibilitam que os estudos primários fiquem restritos ao tema explorado, pois cada estudo pode ser classificado

e pode ser incluído, ou não, no conjunto de estudos para avaliação. Os critérios de inclusão e exclusão definidos nesse método de seleção estão na Tabela 3.4.

Critérios de Inclusão	**Critérios de Exclusão**
Estudos publicados em Ciência da Computação.	Estudos duplicados.
Estudos escritos em inglês.	Estudos classificados como tutoriais, pôsteres, painéis, palestras, mesas redondas, oficinas, teses e dissertações.
Estudos disponíveis em formato eletrônico.	Estudos que não puderem ser localizados.
Estudos publicados em conferências ou revistas.	
Estudos relacionados ao *termo de busca* e, portanto, ao tema da revisão sistemática.	

Tabela 3.4. Critérios de inclusão e exclusão de estudos primários.

As quatro etapas que compõe o método de seleção estão descritas a seguir. Elas consideram os critérios de inclusão e exclusão que foram definidos para a avaliação dos estudos primários. A execução dessas etapas deve ser sequencial, de tal modo que a Etapa 1 processará todos os estudos primários obtidos pela busca nas bases de dados; a Etapa 2 processará apenas os estudos primários que não foram excluídos na Etapa 1, e assim por diante.

Etapa 1: cada um dos estudos primários obtidos deve ser comparado com os demais e deve ser excluído, caso o título e os nomes de todos os autores sejam iguais. Considerando que a quantidade de estudos obtidos será possivelmente elevada, as comparações desta etapa devem ser automatizadas, a fim de agilizar o processo e minimizar a ocorrência de erros.

Etapa 2: todos os estudos primários que não foram excluídos na Etapa 1 devem ser avaliados em seu aspecto geral, e os títulos de todos eles devem ser lidos. Devem ser excluídos os estudos que não estejam no idioma inglês, ou que não pertençam à área da Ciência da Computação, ou que, pela avaliação do título, não estejam relacionados ao tema da revisão sistemática. Caso a leitura do título não permita uma avaliação precisa, o estudo não poderá ser excluído e deverá ser avaliado na próxima Etapa.

Etapa 3: todos os resumos dos estudos primários que não foram excluídos na Etapa 2 devem ser lidos. Estudos cujos resumos indicarem falta de pertinência com o

tema da revisão sistemática devem ser excluídos. Nos casos em que a pertinência com o tema da revisão seja apenas parcial, ou seja, nos casos em que o estudo trate apenas da tarefa de alocação de registradores, ou apenas da tarefa de escalonamento de instruções, sem considerar essas duas tarefas em conjunto e a interdependência entre elas, o estudo também deve ser excluído.

Etapa 4: cada um dos estudos primários deve ser avaliado de acordo com os demais critérios de inclusão e exclusão definidos na Tabela 3.4 e não verificados nas etapas anteriores. Devem ser excluídos os estudos que não puderem ser associados a nenhum dos critérios de inclusão, ou que forem associados a quaisquer critérios de exclusão.

É interessante registrar que nenhuma dessas etapas considera o ano de publicação dos estudos, de tal modo que nenhum estudo primário foi excluído ou incluído da revisão sistemática em função da sua data de publicação.

3.1.5 Extração dos Dados

A extração dos dados dos estudos primários selecionados para análise deverá ser orientada para a obtenção das informações necessárias às respostas das questões de pesquisa. Assim, para cada um dos estudos primários analisados, um resumo deverá ser escrito, em formato livre ou tabular, contendo, no mínimo, o seguinte conjunto de informações: a) título do estudo; b) autores do estudo; c) data de publicação do estudo; d) tipo de abordagem utilizada para a resolução do problema; e) principais características da solução proposta; f) eficiência da solução; g) arquitetura relacionada à solução proposta; h) identificação dos testes realizados; i) ferramentas e softwares utilizados.

3.2 Execução

Nesta fase da revisão sistemática, efetuou-se todo o planejamento descrito na Seção 3.1. Primeiramente, realizou-se o processo de busca dos estudos primários nas bases de dados escolhidas. Depois, foi realizado o processo de seleção dos estudos obtidos, gerando, como resultado, o conjunto de estudos primários para análise. Nas seções que se seguem descrevemos os processos de busca e seleção efetuados e apresentamos o resultado final dessa fase.

3.2.1 Processo de Busca

A busca dos estudos primários foi realizada em cada uma das bases de dados eletrônicas apresentadas na Tabela 3.1. Nelas, o processo de busca é automatizado por um motor de busca existente na interface *web* da própria base. Assim, para a realização da busca, bastou fornecer o *termo de busca* a esse motor, utilizando-se os campos disponíveis na interface, e executar o comando apropriado. Em cada base, os resultados obtidos foram exportados no formato BIBTEX[3], exceto na base *Springer*, que não permite a exportação nesse formato. Para essa base, a exportação foi feita no formato CSV (*Comma Separeted Values*), que é bastante conhecido e pode ser utilizado com a maioria dos editores de texto.

Todo o processo de busca foi realizado entre os dias 19 e 21 de outubro de 2016, de tal modo que estudos primários publicados posteriormente à data final desse período não foram considerados nessa RSL.

A Tabela 3.5 exibe, para cada uma das bases eletrônicas pesquisadas, a quantidade de estudos primários obtidos. Ao todo, 542 estudos primários foram obtidos no processo de busca.

Base de Dados	**Total de Estudos Obtidos**
ACM Digital Library	134
IEEE Xplore	0
Science Direct	313
Scopus	27
Springer	34
Web of Science	34
Total de estudos primários obtidos	**542**

Tabela 3.5. Totais de estudos primários obtidos no processo de busca.

A base *IEEE Xplore* não retornou estudos para a busca realizada com o *termo de busca* definido. Inicialmente, insatisfeitos com esse resultado, novas buscas foram feitas, mudando-se as posições dos conectores lógicos `AND` e `OR` do *termo de busca*, mas mantendo-se as palavras-chave definidas. Somente em alguns casos, as buscas com esses *termos de busca* modificados produziram resultados. Contudo, todos os estudos primários assim obtidos já estavam presentes nos resultados das buscas nas outras

[3]Trata-se de um formato e de um conjunto de ferramentas para descrever e processar listas de referências de documentos. Trabalha de forma especialmente integrada com documentos elaborados com LATEX, que é um sistema para a diagramação de documentos científicos e que se tornou o padrão para a elaboração e publicação desses documentos. LATEX originou-se a partir TEX, que também é um sistema e um padrão para a produção de textos científicos e que foi criado por Donald Knuth no fim da década de 70 na Universidade de Stanford. Mais informações em `http://www.bibtex.org/` e `https://www.latex-project.org/`.

bases, exceto por um único estudo primário. Em função dessa não conformidade no processo, isto é, da necessidade de se alterar, para uma única base, o *termo de busca* definido, e, principalmente, em função da baixa contribuição que essa alteração traria para os resultados da revisão sistemática, o *termo de busca* foi mantido único para todas as bases e o resultado original da busca na base *IEEE Xplore* foi aceito.

3.2.2 Processo de Seleção

Terminado o processo de busca, iniciou-se o processo de seleção dos estudos primários obtidos. A seleção observou as etapas e os critérios de inclusão e exclusão definidos na fase de planejamento. Em cada uma das quatro etapas definidas na Seção 3.1.4, alguns estudos foram excluído e outros foram mantidos. Os estudos que permaneceram após a execução da última etapa compõem o resultado deste processo de seleção e foram considerados na fase de análise da RSL.

A Figura 3.1 exibe, para cada uma das bases de dados, o total de estudos primários que foram mantidos após cada uma das etapas do processo de seleção. A base IEEE não aparece nessa figura, pois não retornou resultados no processo de busca.

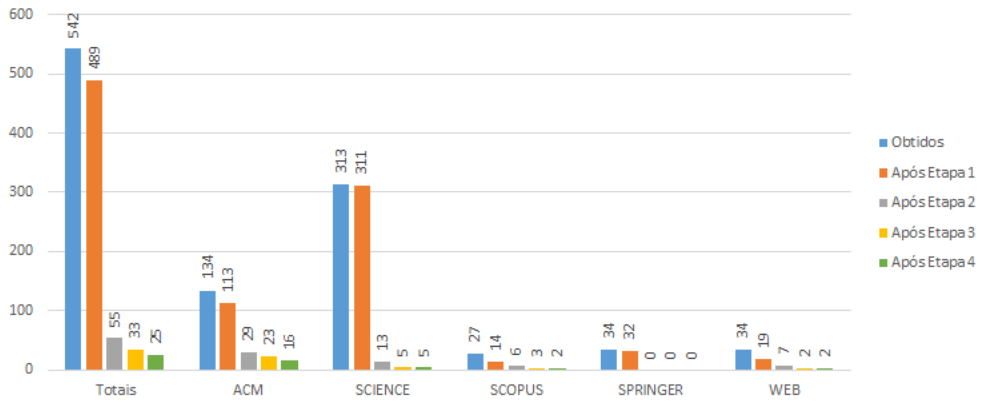


Figura 3.1. Resultados das etapas do processo de análise.

Conforme definido na fase de planejamento, a Etapa 1 foi automatizada. As Etapas 2, 3 e 4 foram realizadas manualmente por três avaliadores. A seguir, apresentamos os detalhes da execução de cada uma das etapas do processo de seleção.

3.2.2.1 Etapa 1

Nesta etapa, o título e os nomes dos autores de cada um dos estudos primários obtidos no processo de busca foram comparados. Os estudos com títulos e nomes de autores iguais aos de algum outro estudo foram excluídos. As comparações realizadas foram

automatizadas, a fim de agilizar o processo e minimizar a ocorrência de erros. A Tabela 3.6 apresenta, para cada uma das bases de dados, o total de estudos obtidos, o total de estudos excluídos e o total de estudos mantidos na revisão, para serem avaliados na Etapa 2. Dos 542 estudos obtidos no processo de busca, 53 foram excluídos e 489 foram mantidos.

	Total de Estudos		
Base de Dados	**Obtidos**	**Excluídos**	**Mantidos**
ACM	134	21	113
SCIENCE	313	2	311
SCOPUS	27	13	14
SPRINGER	34	2	32
WEB	34	15	19
Total de estudos	**542**	**53**	**489**

Tabela 3.6. Resumo dos resultados da Etapa 1.

3.2.2.2 Etapa 2

Nesta etapa, todos os estudos primários mantidos na Etapa 1 foram avaliados em seu aspecto geral, e os títulos de todos eles foram lidos. Estudos publicados em idiomas diferentes do inglês, ou fora da área de Ciência da Computação, ou que não estavam relacionados ao tema da revisão sistemática foram excluídos. Nos casos em que a leitura do título não permitiu uma avaliação precisa, o estudo foi mantido.

Todos os 489 estudos que foram mantidos na etapa anterior foram avaliados. Eles foram distribuídos entre três avaliadores, conforme ilustram as tabelas 3.7 e 3.8.

Base de Dados	**Avaliador**
ACM	Avaliador 1
SCIENCE	Avaliadores 2 e 3
SCOPUS	Avaliadores 2 e 3
SPRINGER	Avaliador 1
WEB	Avaliador 1

Tabela 3.7. Bases de dados por avaliador.

Avaliador	**Total de Estudos**
Avaliador 1	164
Avaliador 2	162
Avaliador 3	163
Total de estudos	**489**

Tabela 3.8. Total de estudos por avaliador.

Cada avaliador foi responsável pela avaliação inicial dos estudos que lhe foram conferidos. Durante essa avaliação, cada estudo recebeu um dos seguintes rótulos:

sim, indicando que o avaliador considerou que o estudo deveria ser mantido na revisão sistemática; *não*, indicando que o estudo deveria ser excluído da revisão sistemática; ou *talvez*, indicando que o avaliador teve dúvidas sobre a classificação do estudo.

Após essa avaliação inicial, os estudos rotulados com *não* foram excluídos da RSL. Todavia, cada um dos estudos rotulados com *sim* ou *talvez* foi então submetido a uma segunda avaliação, feita pelos dois avaliadores que não o avaliaram inicialmente. Nessa segunda avaliação, cada um desses estudos foi novamente rotulado por cada um dos dois avaliadores. Após essa segunda avaliação, os estudos com dois rótulos *não* foram excluídos da revisão sistemática. Os demais estudos foram mantidos, para serem avaliados na Etapa 3.

O resultado dessa etapa é a lista de todos os estudos avaliados juntamente com os seus respectivos rótulos, emitidos por cada um dos avaliadores. Em função do tamanho, a apresentação dessa lista torna-se impraticável. Assim, apresentamos na Tabela 3.9 somente o resumo quantitativo dos resultados das avaliações feitas. Ao todo, dos 489 estudos avaliados, 434 foram excluídos e 55 foram mantidos.

	Total de Estudos				
Avaliador	**Recebidos**	**Excluídos na Primeira Avaliação**	**Excluídos na Segunda Avaliação**	**Excluídos na Etapa**	**Mantidos na Etapa**
Avaliador 1	164	116	12	128	36
Avaliador 2	162	146	0	146	16
Avaliador 3	163	121	39	160	3
Total de estudos	**489**	**383**	**51**	**434**	**55**

Tabela 3.9. Resumo dos resultados da Etapa 2.

3.2.2.3 Etapa 3

Nesta etapa, os 55 estudos primários vindos da Etapa 2 foram avaliados. O resumo de cada um deles foi lido. Estudos cujos resumos indicaram falta de pertinência com o tema da revisão sistemática foram excluídos. Nos casos em que a pertinência com o tema da revisão foi apenas parcial, ou seja, nos casos em que o estudo tratava apenas da tarefa de alocação de registradores, ou apenas da tarefa de escalonamento de instruções, sem considerar essas duas tarefas em conjunto e a interdependência entre elas, o estudo também foi excluído.

Cada um dos estudos, em função dos dados extraídos do seu resumo, recebeu, de cada um dos avaliadores, um dos seguintes rótulos: *sim*, *não* ou *talvez*. Estudos com dois ou mais rótulos *não* foram excluídos. Todos os demais estudos foram mantidos na revisão sistemática. Nesse processo de avaliação, verificou-se a presença de 9 estudos

repetidos. Tais estudos também foram excluídos da revisão e não foram detectados na Etapa 1 em função de pequenas diferenças existentes no título de cada um deles.

A Tabela 3.10 resume os resultados dessa etapa. As tabelas 3.11, 3.12 e 3.13 apresentam, respectivamente, os estudos excluídos nessa etapa, os estudos repetidos, que foram encontrados nessa etapa e que também foram excluídos, e os estudos mantidos nessa etapa. Dos 55 estudos avaliados, 13 foram excluídos e 33 foram mantidos.

	Total de Estudos			
Base de Dados	**Analisados**	**Repetidos**	**Excluídos**	**Mantidos**
ACM	29	2	4	23
SCIENCE	13	0	8	5
SCOPUS	6	2	1	3
SPRINGER	0	0	0	0
WEB	7	5	0	2
Total de estudos	**55**	**9**	**13**	**33**

Tabela 3.10. Resumo dos resultados da Etapa 3.

				Avaliadores		
Número	**Título Resumido**	**Base**	**Ano**	**1**	**2**	**3**
3	Address Register Allocation...Programs	SCIENCE	2003	Não	Não	Não
5	Author Retrospective...Basic Blocks	ACM	2014	Não	Não	Talvez
14	Dynamic Programming...Allocation	SCIENCE	1977	Não	Não	Não
16	Evaluation Of Scheduling...Onto Fpgas	ACM	2004	Não	Não	Talvez
25	Global Approach...Programming	SCIENCE	2003	Não	Não	Não
29	Joint Scheduling And...For Low Power	SCOPUS	1996	Não	Sim	Não
30	Loop Fusion And Reordering...Processors	SCIENCE	2012	Não	Talvez	Não
35	Program Structure As...Allocation	SCIENCE	1993	Não	Não	Não
40	Register Allocation...Loops	ACM	2005	Não	Sim	Não
41	Register Allocation...Systems	SCIENCE	2012	Não	Não	Não
47	Register Assignment...Programs	SCIENCE	1979	Não	Não	Não
49	Rotating Register Allocation...Branches	ACM	2008	Não	Não	Não
50	Scheduling And Allocation...Techniques	SCIENCE	1996	Não	Não	Sim

Tabela 3.11. Lista dos estudos excluídos na Etapa 3.

Número	**Título Resumido**	**Base**	**Ano**	**Estudo Correspondente**
4	An Experimental Study Of...Strategies	ACM	1995	18
17	Experiences With Cooperating...Instruction	WEB	1998	18
19	Experimental Study Of Several...Strategies	SCOPUS	1995	4
20	Fast, Frequency-Based...Instruction	WEB	2008	21
22	Fine-Grain Register...In A Reference	WEB	2010	23
26	Integrated Instruction...Processors	WEB	2006	27
33	Novel Framework Of...Pipelining	SCOPUS	1993	1
43	Register Allocation With...Scheduling	ACM	2010	46
44	Register Allocation With...A New Approach	WEB	2010	46

Tabela 3.12. Lista dos estudos repetidos e também excluídos na Etapa 3.

				Avaliadores		
Número	**Título Resumido**	**Base**	**Ano**	**1**	**2**	**3**
1	A Novel Framework Of...Pipelining	ACM	1993	Sim	Sim	Sim
2	A Systematic Integration...Scheduling	ACM	1999	Talvez	Não	Talvez
6	Code Scheduling And...Basic Blocks	ACM	1988	Sim	Sim	Sim
7	Code Scheduling With...Allocation	SCOPUS	1991	Talvez	Sim	Sim
8	Combined Instruction...Allocation	ACM	1995	Talvez	Não	Sim
9	Combining Register...Scheduling	ACM	1995	Talvez	Sim	Sim
10	Constraint-Based Register...Scheduling	ACM	2012	Sim	Sim	Sim
11	Cooperative Register...Scheduling	ACM	1995	Talvez	Não	Sim
12	Craig: A Practical...Assignment	SCOPUS	1995	Sim	Sim	Sim
13	Design And Implementation...Synthesis	ACM	1994	Talvez	Não	Talvez
15	Evaluating Register...Processors	ACM	1999	Sim	Sim	Sim
18	Experiences With Cooperating...Scheduling	ACM	1998	Sim	Sim	Sim
21	Fast, Frequency-Based,...Scheduling	ACM	2008	Sim	Sim	Sim
23	Fine-Grain Register...Flow	ACM	2010	Sim	Não	Sim
24	Genetic Instruction...Allocation	ACM	2004	Sim	Sim	Sim
27	Integrated Instruction...Processors	ACM	2006	Sim	Sim	Sim
28	Integrating Register...For Riscs	ACM	1991	Sim	Sim	Talvez
31	Minimizing Schedule...Systems	ACM	2011	Sim	Sim	Sim
32	Minimum Register Instruction...Processors	ACM	2000	Sim	Sim	Sim
34	On Minimizing Register...Dependencies	ACM	2010	Não	Sim	Sim
36	Register Allocation...Architectures	ACM	1987	Talvez	Talvez	Talvez
37	Register Allocation...In Unison	ACM	2016	Sim	Sim	Sim
38	Register Allocation...Loop Unrolling	ACM	2016	Sim	Sim	Sim
39	Register Allocation...Processor	SCIENCE	2015	Sim	Sim	Sim
42	Register Allocation...Scheduling	SCOPUS	1995	Sim	Sim	Sim
45	Register Allocation...Vliw-Architectures	ACM	2010	Sim	Sim	Sim
46	Register Allocation...A New Approach	WEB	2010	Sim	Não	Sim
48	Resource Spackling...Schedulers	WEB	1994	Sim	Sim	Sim
51	Scheduling Expression...Register Need	SCIENCE	1998	Sim	Talvez	Não
52	Scheduling With Register...Architectures	SCIENCE	1994	Sim	Talvez	Talvez
53	Unification Of Register...Architectures	ACM	1996	Talvez	Não	Sim
54	Using Integer Linear...Processors	SCIENCE	1997	Sim	Sim	Sim
55	Variable Assignment...Memory	SCIENCE	2011	Talvez	Sim	Talvez

Tabela 3.13. Lista dos estudos mantidos na Etapa 3.

3.2.2.4 Etapa 4

Nesta etapa, os estudos primários mantidos na Etapa 3 foram avaliados de acordo com os critérios de inclusão e exclusão que foram definidos na Tabela 3.4 e que não foram verificados nas etapas anteriores. Foram excluídos os estudos que não puderam ser associados a nenhum dos critérios de inclusão, ou que foram associados a quaisquer critérios de exclusão. Basicamente, um critério de inclusão e dois critérios de exclusão precisaram ser considerados nesta etapa. O critério de inclusão foi a disponibilidade dos estudos em formato digital. Os critérios de exclusão foram a classificação e a disponibilidade dos estudos.

Após as avaliações desta etapa, dos 33 estudos recebidos da etapa anterior, 8

estudos foram excluídos e 25 foram selecionados para a fase de análise da revisão sistemática. A Tabela 3.14 apresenta a lista dos estudos excluídos com seus respectivos motivos de exclusão. Os estudos selecionados para a fase de análise da revisão sistemática, que são o resultado final deste processo de seleção, estão apresentados na Tabela 3.15 da Seção 3.2.3.

Número	Título Resumido	Base	Ano	Motivo
2	A Systematic Integration...Scheduling	ACM	1999	Tese de doutorado
7	Code Scheduling With...Allocation	SCOPUS	1991	Indisponível
8	Combined Instruction...Allocation	ACM	1995	Indisponível
11	Cooperative Register...Scheduling	ACM	1995	Tese de doutorado
13	Design And Implementation...Synthesis	ACM	1994	Tese de doutorado
27	Integrated Instruction...Processors	ACM	2006	Indisponível
36	Register Allocation...Architectures	ACM	1987	Tese de doutorado
53	Unification Of Register...Architectures	ACM	1996	Tese de doutorado

Tabela 3.14. Lista dos estudos excluídos na Etapa 4.

3.2.3 Resultado Final

Com o término da Etapa 4 e, portanto, do processo de seleção, a fase de execução da RSL foi concluída. Como resultado, dentre os 542 estudos primários avaliados, encontramos 25 estudos relacionados ao tema de pesquisa da revisão sistemática, ou seja, estudos que tratam das tarefas de alocação de registradores e escalonamento de instruções, bem como da interdependência entre elas.

A Tabela 3.15 apresenta esses 25 estudos, que são o conjunto de estudos selecionados para a fase de análise da revisão sistemática. Nessa tabela, os estudos estão ordenados pela ordem crescente da sua data de publicação. O estudo mais antigo foi publicado em 1988 e os mais recentes em 2016.

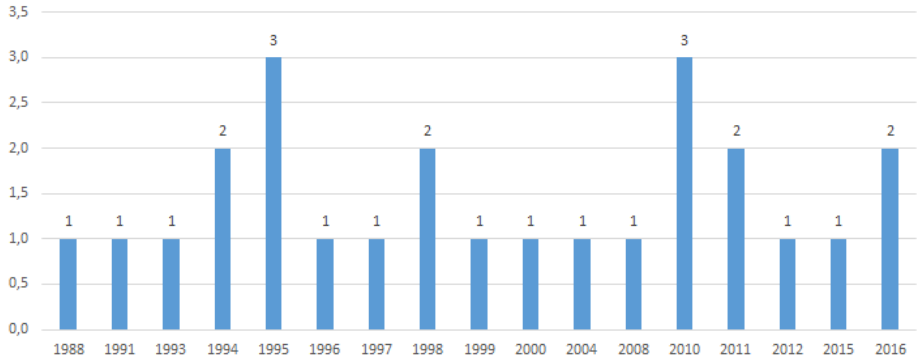


Figura 3.2. Estudos primários para análise por ano de publicação.

A Figura 3.2 apresenta o total de estudos primários para análise por ano de publicação. As Figuras 3.3 (a) e (b) exibem os totais de estudos por base de dados e por década de publicação respectivamente. É interessante observar que entre os anos de 2010 e 2016, foram publicados um total de 9 artigos, uma média aproximada de 1 artigo por ano.

Estudo	**Título**	**Base**	**Ano**
1	Code Scheduling and Register Allocation in Large Basic Blocks	ACM	1988
2	Integrating Register Allocation and Instruction Scheduling for RISCs	ACM	1991
3	A Novel Framework of Register Allocation for Software Pipelining	ACM	1993
4	Resource Spackling - A Framework for Integrating Register Allocation in Local and Global Schedulers	WEB	1994
5	Scheduling with Register Constraints for DSP Architectures	SCIENCE	1994
6	Combining Register Allocation and Instruction Scheduling	ACM	1995
7	CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment	SCOPUS	1995
8	Register Allocation Sensitive Region Scheduling	SCOPUS	1995
9	Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-issue Processors	SCIENCE	1997
10	Experiences with Cooperating Register Allocation and Instruction Scheduling	ACM	1998
11	Scheduling Expression DAGs for Minimal Register Need	SCIENCE	1998
12	Evaluating Register Allocation and Instruction Scheduling Techniques in Out-Of-Order Issue Processors	ACM	1999
13	Minimum Register Instruction Scheduling: A New Approach for Dynamic Instruction Issue Processors	ACM	2000
14	Genetic Instruction Scheduling and Register Allocation	ACM	2004
15	Fast, Frequency-based, Integrated Register Allocation and Instruction Scheduling	ACM	2008
16	Fine-Grain Register Allocation and Instruction Scheduling in a Reference Flow	ACM	2010
17	On Minimizing Register Usage of Linearly Scheduled Algorithms with Uniform Dependencies	ACM	2010
18	Register Allocation with Instruction Scheduling for VLIW-architectures	ACM	2010
19	Register Allocation with Instruction Scheduling: A New Approach	WEB	2010
20	Minimizing Schedule Length via Cooperative Register Allocation and Loop Scheduling for Embedded Systems	ACM	2011
21	Variable Assignment and Instruction Scheduling for Processor with Multi-module Memory	SCIENCE	2011
22	Constraint-Based Register Allocation and Instruction Scheduling	ACM	2012
23	Register Allocation for Fine Grain Threads on Multicore Processor	SCIENCE	2015
24	Register Allocation and Instruction Scheduling in Unison	ACM	2016
25	Register Allocation and Promotion Through Combined Instruction Scheduling and Loop Unrolling	ACM	2016

Tabela 3.15. Resultado final da fase de execução. Estudos primários selecionados para a fase de análise.

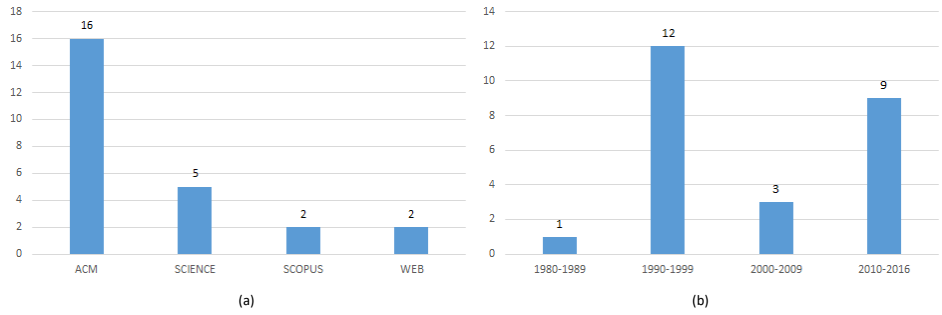


Figura 3.3. Estudos primários para análise: (a) por base de dados; (b) por década de publicação.

3.3 Análise

Finalizada a fase de execução, iniciou-se a fase de análise dos estudos primários selecionados. Primeiramente, esses estudos foram lidos e resumidos, extraindo-se de cada um deles os dados necessários para se responder as questões de pesquisa da RSL. Posteriormente, com base nos dados extraídos, essas questões de pesquisa foram respondidas.

3.3.1 Extração dos Dados

Nesta seção, apresentamos os resumos dos estudos primários selecionados para análise. Esses resumos foram escritos em formato livre e feitos conforme o planejamento indicado na Seção 3.1.5. A apresentação dos resumos segue a mesma ordem utilizada para a listagem dos estudos na Tabela 3.15.

1. Code Scheduling and Register Allocation in Large Basic Blocks

Goodman & Hsu [1988] exemplificam didaticamente o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções e propõem dois métodos para solucioná-lo.

O primeiro deles é um método de escalonamento que combina duas técnicas: CSP (*Code Scheduling for Pipelined processors*), que busca reduzir atrasos no *pipeline*, e CSR (*Code Scheduling to minimize Registers usage*), que visa minimizar a utilização dos registradores. Esse método sequencia as instruções mantendo o número de registradores disponíveis sob controle, a fim de evitar derramamentos para a memória. O escalonador utiliza CSP na maior parte do tempo. Caso o número de registradores disponíveis fique abaixo de um determinado limite, o escalonador aplica a técnica CSR, voltando à técnica CSP tão logo esse número se torne maior ou igual ao limite estabelecido.

Como segunda proposta para solucionar o problema da interdependência, os autores apresentam um método de alocação de registradores, chamado *DAG-Driven Register Allocation*. Esse método utiliza o grafo de interferência como estrutura de dados e busca minimizar a altura desse grafo. Por definição, essa altura é o comprimento do caminho mais longo que existe no grafo. A alocação dos registradores é feita controlando-se o valor dessa altura, uma vez que, quanto maior for esse valor, menos eficiente será o escalonamento do código, ou ainda, menos possibilidades de paralelismo existirão nesse escalonamento. Para o controle dessa altura duas estratégias são propostas: *Free WAR Dependencies* e *Balancing the Growth of the DAG*.

Pelos testes realizados, o método de escalonamento mostrou-se mais flexível e também mais agressivo, forçando o derramamento de valores para a memória para favorecer o sequenciamento das instruções. O método de alocação, por sua vez, mostrou-se conceitualmente mais simples. Ambos os métodos trataram o problema localmente, isto é, dentro de um único bloco básico. Nas simulações realizadas, considerando blocos básicos grandes, ambos os métodos geraram códigos mais eficientes do que as técnicas *prepass* e *postpass* convencionais.

2. Integrating Register Allocation and Instruction Scheduling for RISCs

Bradlee et al. [1991] tratam do impacto das fases de alocação de registradores e escalonamento de instruções na geração de código para sistemas com um único processador RISC que suporta operações multiciclo. Os autores buscam responder duas questões: ❶ se a falta de comunicação entre a fase de alocação de registradores e a fase de escalonamento de instruções produz códigos ineficientes; ❷ caso produza, quão integrada devem ser essas duas funções.

Para responderem essas questões, Bradlee et al. apresentam e comparam os resultados de três estratégias de geração de código: ❶ Postpass [Hennessy & Gross, 1983; Gibbons & Muchnick, 1986], na qual a alocação de registradores é feita antes do escalonamento de instruções e de forma totalmente separada; ❷ uma variação do IPS (*Integrated Prepass Scheduling*) [Goodman & Hsu, 1988], na qual o escalonamento de instruções é realizado antes da alocação de registradores, mas utilizando heurísticas que consideram certas restrições impostas pelo alocador; ❸ RASE (*Register Allocation with Schedule Estimates*) [Bradlee et al., 1991], criada pelos autores do artigo, na qual há integração das duas fases por meio da introdução de uma etapa de pré-escalonamento, que estima custos de escalonamento e os entrega ao alocador de registradores, a fim de que ele balanceie o uso dos registradores entre o objetivo de evitar atrasos no *pipeline* e o de reduzir acessos à memória.

Os códigos gerados pelas estratégias IPS e RASE foram, em média, 12% mais rápidos do que aqueles gerados com a estratégia Postpass, donde os autores concluem que a execução totalmente separada das fases de alocação e escalonamento produz códigos ineficientes e que a integração dessas duas fases é necessária. Todavia, verificaram que, embora em RASE a integração entre as duas fases seja maior, ela não apresentou resultados significativamente melhores do que a segunda estratégia, exceto em poucos programas com muitos blocos básicos grandes, concluindo que a estratégia IPS é a melhor escolha, uma vez que é conceitualmente mais simples e implica menores tempos de compilação.

3. A Novel Framework of Register Allocation for Software Pipelining

Ning & Gao [1993] buscam minimizar o tempo de execução de laços. Propõem um método no qual a alocação de registradores é tratada como uma restrição para o escalonamento de instruções. O método é composto por duas etapas: a primeira determina um escalonamento com tempo ótimo de execução capaz de utilizar uma quantidade mínima de registradores; a segunda aloca os registradores de acordo com o escalonamento definido. Para a realização da primeira etapa, pseudoregistradores são atribuídos às variáveis existentes no laço e armazenados em um *buffer*. Busca-se então identificar um escalonamento ótimo que minimiza o tamanho desse *buffer*, considerando-se as diversas iterações do laço. Os escalonamentos são realizados com *software pipelinig*[4] e o problema é formulado como um problema de otimização, utilizando-se *programação inteira*. Segundo os autores, transformando esse problema em um *problema de fluxo de custo mínimo*, uma solução em tempo polinomial pode ser encontrada. Para a realização da segunda etapa, isto é, para a alocação dos registradores que restaram no *buffer* após a definição do escalonamento ótimo, qualquer algoritmo de alocação pode ser utilizado. No estudo, os autores indicam a utilização de um método baseado em *grafos de intervalo cíclico* [Hendren et al., 1992]. Os resultados obtidos por esse método foram comparados aos resultados obtidos utilizando-se o método descrito em Rau et al. [1992]. Uma redução de 25% na quantidade de registradores alocados foi observada, sem comprometer o ganho de velocidade obtido com o escalonamento.

4. Resource Spackling - A Framework for Integrating Register Allocation in Local and Global Schedulers

Berson et al. [1994] apresentaram o *framework Resource Spacking* para a realização

[4]Técnica para a otimização de laços, semelhante à técnica de *pipelinig*, mas que, ao contrário desta última, não é executada pelo hardware, e sim pelo compilador [Aho et al., 2006].

das tarefas de alocação de registradores e escalonamento de instruções de forma integrada. Esse *framework* foi construído com base no paradigma de *Medição* e *Redução*. Essa técnica mede os requisitos dos recursos demandados pelo programa e, a partir das medições feitas, determina as regiões do programa em que esses recursos estão sendo subutilizados (*resource holes*), ou sobrecarregados (*excessive sets*). Objetivando então aumentar a utilização dos recursos nos *resource holes*, as instruções do código são movimentadas local, ou globalmente. *Resource Spacking* foi implementado em um compilador experimental, denominado *pdgcc*, capaz de gerar código intermediário na forma de PDGs (*Program Dependence Graphs*) e realizar análises de dependências. Os recursos definidos no *framework* foram registradores e unidades funcionais e o código foi gerado para duas arquiteturas VLIW, cada uma delas com um certo número desses recursos. Foram realizados testes utilizando seis procedimentos da versão C do *benchmark* linpack. Em todos eles, os autores relataram que o paralelismo existente foi corretamente explorado e que o escalonamento realizado tornou o código mais eficiente.

5. Scheduling with Register Constraints for DSP Architectures

Em Depuydt et al. [1994], a interação entre alocação de registradores e escalonamento de instruções é abordada visando a geração de código para processadores com arquitetura ASIP (*Application-Specific Instruction-set Processors*), que são processadores programáveis, muitas vezes utilizados em sistemas embarcados para aplicações DSP (*Digital Signal Processing*) de tempo real. A arquitetura ASIP possui as seguintes características: ❶ possui várias unidades funcionais paralelas; ❷ os vários arquivos de registradores estão distribuídos; ❸ realiza suas operações a partir dos registradores; ❹ é programável.

A técnica proposta para o acoplamento das fases de alocação de registradores e escalonamento de instruções assemelha-se àquela apresentada em Rimey [1989], diferindo por considerar todo o espaço de solução para o escalonamento. Ela utiliza uma técnica, baseada em *retiming* [Leiserson et al., 1983], para calcular em tempo polinomial o número máximo de sinais vivos no grafo HDSFG (*Hierarchical Decorated Signal Flow Graph*), que é um grafo de fluxo de controle e dados. Utilizando o HDSFG, o custo máximo dos registradores é reduzido antes da execução do escalonamento, livrando-o das restrições relacionadas à alocação de registradores e permitindo que ele aproveite ao máximo o *pipeline* para reduzir o tempo de execução da aplicação.

Os autores afirmaram a eficiência da técnica cooperativa proposta, realizando experimentos em uma ferramenta de desenvolvimento de protótipos denominada TRON. Contudo, resultados comparativos entre o método desenvolvido e outros métodos não

foram apresentados. Além disso, os autores afirmam que a metodologia utilizada não garante o escalonamento ótimo.

6. Combining Register Allocation and Instruction Scheduling

Motwani et al. [1995] definiram o modelo CRISP (Combined Register allocation and Instruction Scheduling Problem) que trata as tarefas de alocação e escalonamento dentro de um bloco básico como um único problema de otimização. A função de custo desse problema, função objetivo a ser minimizada, é o tempo de execução da última instrução do bloco.

CRISP necessita que o código intermediário esteja na forma SSA, de tal modo que cada uma das variáveis do bloco seja definida uma única vez. Ele utiliza o algoritmo baseado em heurística (α, β)-*Combined Heuristic*, que foi definido pelos próprios autores. Nesse algoritmo, α e β são parâmetros utilizados para controlar a pressão sobre os registradores e o paralelismo das instruções respectivamente. Inicialmente, esse algoritmo classifica os pseudoregistradores e as instruções do bloco utilizando funções indicadas no próprio modelo. A partir dessas classificações, um *rank* final de instruções é construído conforme a heurística definida e, posteriormente, escalonado com um algoritmo guloso de escalonamento de lista.

Para avaliar o modelo, os autores criaram instâncias teóricas do problema da interdependência, representadas por grafos de escalonamento. Essas instâncias foram então utilizadas em simulações teóricas considerando arquiteturas com quatro, oito e dezesseis registradores. Cada uma dessas três arquiteturas possuía um único *pipeline* de dois estágios. Os experimentos realizados compararam a solução do modelo *CRISP* com a solução gerada pela execução ordenada das tarefas de alocação e escalonamento, tanto em uma abordagem *prepass*, quanto em uma abordagem *postpass*. Os resultados obtidos indicaram que a utilização do modelo *CRISP* resultou em um ganho de performance de até 21% em relação à execução ordenada das tarefas.

7. CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment

Brasier et al. [1995] propuseram o *framework* CRAIG (*Combining Register Assignment Interference Graphs*) para a realização cooperativa das tarefas de alocação e escalonamento. CRAIG foi inspirado no trabalho realizado por Pinter [Pinter, 1993] e utiliza o grafo de interferência como sua principal estrutura de dados. Ele define um *custo de escalonamento* para retratar os objetivos do gerador de código. Esse custo é uma heurística que considera tanto a eficiência do escalonamento, quanto a pressão sobre

os registradores, e está relacionado ao grafo de interferência que deriva do código intermediário sob análise.

Para solucionar o problema, o *framework* utiliza inicialmente a abordagem *prepass*, escalonando as instruções e, posteriormente, alocando os registradores. Se o custo de escalonamento assim obtido for aceitável, então o código gerado será a solução do problema. Caso contrário, se esse custo for muito alto, CRAIG armazena o grafo de interferência gerado nessa abordagem, descarta o código obtido e executa a abordagem *postpass*, alocando os registradores primeiramente e escalonando as instruções em seguida. Caso o custo de escalonamento obtido com essa abordagem permaneça alto, o algoritmo assume que nada melhor pode ser feito e considera que o código gerado dessa maneira é a solução para o problema. Caso contrário, se o custo de escalonamento não for muito alto, o método assume que a alocação de registradores gerou muitas falsas dependências e que, portanto, o sequenciamento de instruções pode ser melhorado. Para isso, o algoritmo adiciona ao grafo de interferência uma das arestas do grafo gerado pela abordagem *prepass* e executa novamente as tarefas de alocação e escalonamento. Esse processo é repetido, inserindo-se novas arestas no grafo e executando-se novamente as tarefas de alocação e escalonamento, até que o custo de escalonamento se torne muito alto. Quando isso ocorrer, o código gerado é considerado a solução do problema. Dessa forma, CRAIG utiliza um mecanismo para diminuir falsas dependências, aumentando a liberdade do escalonamento mesmo que isso acarrete o derramamento de variáveis para a memória.

Uma instância do *framework* foi implementada no compilador ROCKET, um compilador C capaz de gerar código para arquiteturas superescalares LIW. Um modelo de arquitetura LIW foi definido nesse compilador e o código foi gerado para esse modelo, não para uma máquina real. Um simulador URM (*Unlimited Register Machine*) foi utilizado para executar o código e os testes realizados indicaram que o código gerado por CRAIG foi mais eficiente do que o código gerado com a execução separada das tarefas de alocação de registradores e escalonamento de instruções.

8. Register Allocation Sensitive Region Scheduling

Norris & Pollock [1995] descrevem a estratégia que utilizaram para modificar a técnica de escalonamento global de instruções conhecida como escalonamento de regiões[5] (*region scheduling*), a fim de que ela considere as necessidades de uma posterior fase de alocação de registradores. O novo escalonador criado a partir dessa modificação

[5]Essa técnica também é conhecida como escalonamento regional, ou escalonamento baseado em regiões [Aho et al., 2006].

foi denominado RASER (*Register Allocation Sensitive Region Scheduling*). Norris & Pollock construíram tanto o escalonador baseado em regiões convencional, quanto o escalonador RASER e compararam os resultados obtidos com cada um deles.

O escalonamento regional utiliza como estrutura de dados o PDG (*Program Dependence Graph*) e tenta criar regiões com quantidades iguais de paralelismo de baixa granularidade. RASER trabalha como um escalonador regional e também utiliza o PDG, contudo tenta prevenir os derramamentos para a memória que ocorrerão na fase de alocação de registradores. Para isso, ele utiliza uma técnica de escalonamento denominada Redução de Valores Vivos (*Live Value Reduction*), que foi definida no próprio artigo. Assim como ocorre com um escalonador regional convencional, RASER estima o paralelismo existente em cada uma das regiões do PDG e, posteriormente, escalona as instruções de cada uma dessas regiões, utilizando o método IPS. Contudo, RASER também constrói uma lista com as regiões que possuem uma quantidade de variáveis vivas maior do que o número de registradores que estão disponíveis na arquitetura alvo. As variáveis dessas regiões são aquelas que poderão ser derramadas para a memória durante a fase de alocação de registradores, por isso, RASER tenta reduzir o número de variáveis nessas regiões, aplicando a técnica de redução de valores vivos.

RASER foi implementado no compilador PDGCC e os autores verificaram que ele gerou códigos com menos derramamentos para a memória em todos os experimentos realizados. Também gerou códigos mais rápidos, reduzindo o número de ciclos executados, principalmente, nos cenários com poucos registradores disponíveis. Entretanto, RASER não se mostrou eficiente para maximizar as possibilidades de paralelismo. Sendo assim, uma vez que a importância da reordenação de instruções para aproveitar as oportunidades de paralelismo aumenta, à medida que o número de registradores cresce, RASER mostrou um baixo desempenho nos cenários com muitos registradores disponíveis.

9. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-issue Processors

Em Chang et al. [1997], as tarefas de alocação de registradores e escalonamento de instruções são tratadas como um problema de otimização matemática e sua solução simultânea é alcançada utilizando-se Programação Linear Inteira (*ILP – Integer Linear Programming*).

Chang et al. definiram dois problemas de otimização: o problema NRS, no qual o escalonamento de instruções e a alocação de registradores em um bloco básico são realizados *sem* a possibilidade de derramamento de variáveis para a memória; e o

problema RS, no qual o escalonamento e a alocação em um bloco básico são realizados *com* a possibilidade de derramamento. Para a resolução desses dois problemas, duas otimizações foram consideradas: ❶ otimização de tempo: a partir de um número fixo de registradores livres, busca-se o menor número de ciclos necessários para executar as instruções; ❷ otimização de registradores: a partir de um número máximo e fixo de ciclos de execução, tenta-se encontrar o menor número de registradores necessários. Dentre as restrições impostas aos dois problemas, estão a utilização de um modelo de arquitetura similar à arquitetura RISC e a limitação dos intervalos de vida das variáveis a um único bloco básico, ou seja, o escalonamento proposto é local.

Dois exemplos foram utilizados para os testes de cada um dos dois problemas. Todos os testes foram realizados em uma estação de trabalho SPARC-10, utilizando-se o pacote LINDO [LINDO Systems, 2003] para a solução dos problemas ILP. Após a formulação dos problemas para os exemplos de teste, verificou-se que os problemas RS, que permitem o derramamento de variáveis para a memória, envolvem aproximadamente quatro vezes mais variáveis e inequações do que os problemas NRS, que não permitem o derramamento de variáveis para a memória. Por esse motivo, não foi possível encontrar resultados para o problema RS formulado para o exemplo mais complexo proposto. Além disso, após os testes, verificou-se que o tempo necessário para se encontrar a solução do problema RS formulado para o exemplo mais simples foi cerca de 60 vezes maior do que o tempo necessário para se encontrar a solução do problema NRS formulado para esse mesmo exemplo. Apesar do longo tempo para a resolução dos problemas ILP, soluções ótimas de escalonamento foram obtidas, quando um número muito pequeno de registradores foi fixado. Considerando os resultados, os autores destacaram que a abordagem proposta poderia ser interessante para aplicações de síntese de alto nível (HLS - *High-level synthesis*[6]) de ASICs (*Application Specific Integrated Circuits*) e em projetos de processadores embarcados, domínios nos quais a qualidade do código é mais importante do que o tempo de compilação.

10. Experiences with Cooperating Register Allocation and Instruction Scheduling

O objetivo de Norris & Pollock [1998] foi investigar as vantagens e desvantagens da cooperação entre a tarefa de alocação global de registradores e as tarefas de escalonamento local e global de instruções. Para isso, eles propuseram uma abordagem cooperativa composta pelas seguintes três fases: escalonamento global de instruções, alocação glo-

[6]HLS é um processo automatizado de projeto de hardware capaz de interpretar um algoritmo que descreve um comportamento desejado (uma característica), criando um hardware que implementa esse comportamento.

bal de registradores e escalonamento local de instruções. A estratégia dessa abordagem é fazer com que cada uma dessas fases leve em consideração as necessidades da fase posterior. Dessa forma, o escalonamento global é sensível às necessidades da alocação global, que, por sua vez, leva em consideração as ações do escalonamento local.

A fase de escalonamento global de instruções é realizada pelo escalonador RASER, descrito em Norris & Pollock [1995] e comentado anteriormente nesta seção.

As fases de alocação global de registradores e escalonamento local de instruções são realizadas por um único processo, denominado SSG (*Scheduler Sensitive Global Register Allocator*). Esse processo baseia-se nas abordagens propostas em Goodman & Hsu [1988], que foram apresentadas anteriormente nesta seção. Os requisitos considerados para o projeto do SSG foram os seguintes: ❶ explorar as vantagens do processo de alocação global de registradores; ❷ invocar o escalonador de instruções uma única vez; ❸ evitar a complexidade de realizar as tarefas de alocação e escalonamento simultaneamente; ❹ fazer todas as variáveis competirem igualmente por registradores, a fim de evitar o problema de dividir os registradores entre variáveis locais e globais.

SSG aloca os registradores utilizando o algoritmo *Optimist Coloring*, desenvolvido por Briggs et al. [1992]. Contudo, em cada uma das etapas desse algoritmo, os objetivos da subsequente fase de escalonamento são levados em consideração, permitindo que os códigos gerados em virtude dos derramamentos de variáveis para a memória sejam escalonados juntamente com as demais instruções do programa durante a execução normal do escalonador.

Os códigos desenvolvidos foram implementados no compilador pdgcc. Cinco diferentes cenários de cooperação entre as tarefas de alocação e escalonamento foram produzidos, e testes foram realizados para determinar qual desses cenários proporcionaria o melhor *speedup* em relação a uma abordagem *postpass* convencional. Segundo os autores, o maior *speedup* foi alcançado fazendo-se o escalonamento global de instruções, de forma cooperativa ou não cooperativa, e logo após a alocação global de registradores de forma cooperativa com o escalonamento local de instruções.

11. Scheduling Expression DAGs for Minimal Register Need

Kessler [1998] apresenta um novo algoritmo de escalonamento local que utiliza programação dinâmica e técnicas de reordenação para gerar um escalonamento ótimo de instruções em blocos básicos representados por um DAG, objetivando minimizar o número de registradores. Segundo Kessler, o algoritmo mostrou-se eficiente não apenas para DAGs de pequeno tamanho, mas também para DAGs médios, com até 50 vértices, que representam os blocos básicos de maior parte das aplicações reais.

Ao contrário dos algoritmos usuais para a solução do problema, que possuem complexidade fatorial em relação ao número de instruções, o algoritmo proposto possui complexidade exponencial em relação ao número de instruções no pior caso. Além disso, conforme indicado pelo autor, várias possibilidades de paralelização do algoritmo podem ser exploradas, reduzindo o seu tempo de execução.

12. Evaluating Register Allocation and Instruction Scheduling Techniques in Out-Of-Order Issue Processors

O trabalho de Valluri & Govindarajan [1999] relaciona-se às arquiteturas superescalares com capacidade de escalonamento dinâmico e emissão de instruções fora de ordem. O objetivo dos autores é responder as duas seguintes questões: ❶ as técnicas de compilação que tratam as tarefas de alocação de registradores e escalonamento de instruções e que produzem códigos com melhor desempenho para as arquiteturas que emitem instruções em ordem também produzem códigos com melhor desempenho para as arquiteturas que emitem instruções fora de ordem? ❷ Para as arquiteturas que emitem instruções fora de ordem seria melhor executar a tarefa de escalonamento antes, ou depois da tarefa de alocação de registradores? Em outras palavras: seria melhor o compilador explorar o paralelismo de instruções, ou evitar o derramamento de valores para a memória?

Para responderem essas perguntas, os autores compararam as quatro seguintes abordagens: método *prepass* convencional descrito em Goodman & Hsu [1988]; método *postpass* convencional descrito em Gibbons & Muchnick [1986]; método *prepass* integrado definido em Goodman & Hsu [1988]; e método *postpass* cooperativo definido em Pinter [1993].

Os códigos de cada uma dessas abordagens foram implementados nos *frameworks* SUIF e Machine SUIF. Um conjunto de 22 *benchmarks* foi utilizado para testar cada uma das quatro abordagens consideradas. As compilações dos *benchmarks* geraram códigos na linguagem Alpha Assembly, e a execução desses códigos foi simulada com o software Simplescalar [Burger et al., 1996], que também foi utilizado para coletar os dados de desempenho de cada simulação.

De acordo com os experimentos realizados, os autores concluíram que: ❶ as abordagens não convencionais, que tratam as tarefas de alocação e escalonamento de forma integrada, ou cooperativa, apresentaram ganhos inexpressivos em relação às abordagens convencionais; ❷ as abordagens *postpass* mostraram melhores resultados em *benchmarks* com alta pressão por registradores; ❸ para as arquiteturas superescalares com iniciação de instruções fora de ordem, o foco das abordagens deve ser a minimização

dos derramamentos de valores para a memória.

13. Minimum Register Instruction Scheduling: A New Approach for Dynamic Instruction Issue Processors

Em função do surgimento das arquiteturas superescalares, capazes de escalonar instruções dinamicamente, iniciando-as fora de ordem, Govindarajan et al. [2000] buscam encontrar escalonamentos que utilizem o menor número possível de registradores. Segundo os autores, tais escalonamentos são os que reduzem as possibilidades de derramamento de variáveis para a memória e por isso, ainda que impliquem a perda de uma certa quantidade de paralelismo de instruções em tempo de compilação, são eles os que proporcionam melhor desempenho nas arquiteturas superescalares. Em resumo, os autores se propõem responder a seguinte questão: *"dado um grafo de dependência de dados G, é possível determinar um escalonamento S para G que utilize o menor número possível de registradores?"*

Para responderem essa pergunta, os autores elaboraram uma abordagem heurística que estima o menor número de registradores requeridos pelo DDG (*Data Dependence Graph*), grafo de dependência de dados, considerando todas as possibilidades de escalonamento desse grafo. A esse número mínimo de registradores, os autores deram o nome de MRB (*Minimum Register Bound*). A abordagem elaborada identifica as correntes[7] do DDG que podem compartilhar os mesmos registradores e, a partir disso, constrói o *grafo de interferência de correntes*, cujos vértices representam as correntes e as arestas conectam as correntes que, necessariamente, estarão sobrepostas em qualquer um dos escalonamentos possíveis do DDG. De acordo com Govindarajan et al., o número cromático do grafo de interferência de correntes, que pode ser computado com uma técnica de coloração qualquer, seria a melhor estimativa para o MRB.

Govindarajan et al. propõem ainda um algoritmo de escalonamento de lista que tenta escalonar o código utilizando o MRB como referência. Todavia, os autores destacam que, devido a certas heurísticas utilizadas no algoritmo, um número de registradores maior do que o MRB pode ser necessário para a realização do escalonamento.

A abordagem apresentada não foi implementada pelos autores e, portanto, nenhum resultado experimental foi apresentado no trabalho.

14. Genetic Instruction Scheduling and Register Allocation

Kri & Feeley [2004] utilizam *algoritmos genéticos* para a realização conjunta das tarefas

[7] Os autores do artigo preferiram utilizar o termo *corrente* (*chain*) para se referirem a um *caminho* (*path*) dentro de um grafo. Esse é um termo existente na literatura, mas muito raramente utilizado.

de alocação de registradores e escalonamento de instruções. O objetivo da modelagem realizada foi a minimização do tempo de execução do código gerado. Assim, a função de avaliação (*fitness function*[8]) escolhida foi o tempo real de execução do código. A modelagem do problema considerou cada programa objeto como a concatenação do alocador de registradores e do escalonador de instruções, sendo que cada um deles foi modelado considerando suas dependências em relação ao outro. Devido à função de avaliação escolhida, os resultados obtidos com essa abordagem mostraram um comportamento mais homogêneo, quando comparados aos resultados obtidos com a utilização de heurísticas. Isso porque os efeitos reais da otimização na arquitetura alvo puderam ser considerados, incluindo os efeitos indiretos, que normalmente são difíceis de serem considerados pelas heurísticas. Além disso, ainda devido à escolha da função de avaliação, a modelagem precisou de poucas informações sobre a arquitetura alvo, o que se mostra como uma vantagem, devido à crescente complexidade das arquiteturas de máquina e a possibilidade de simplificação dos compiladores.

15. Fast, Frequency-based, Integrated Register Allocation and Instruction Scheduling

Cutcutache & Wong [2008] descrevem em seu trabalho uma abordagem integrada para a redução do tempo de compilação dos programas. Um algoritmo é utilizado para dividir o tempo de vida das variáveis conforme as regiões do programa em que ela se encontra. Os testes realizados mediram o tempo de compilação e o desempenho dos algoritmos de escalonamento e alocação do compilador *OpenIMPACT*. Segundo os autores, os resultados indicam que a utilização do algoritmo reduziu em 50% o tempo tempo de execução dessas tarefas.

16. Fine-Grain Register Allocation and Instruction Scheduling in a Reference Flow

Kim & Lee [2010] propõem uma nova técnica de alocação que combina as vantagens das abordagens de coloração de grafos [Chaitin et al., 1981] e da técnica *fine-grain*, pela qual a alocação de um registrador para uma variável é feita a cada vez que ela é referenciada, e não uma única vez [Kolte & Harrold, 1993]. Os autores apresentam também uma nova técnica, denominada IRIS (Integrated Register Allocation and Instruction Scheduling), para a integração das tarefas de alocação e escalonamento. Os autores relatam em seu

[8]Um tipo especial de função objetivo utilizada para indicar o quão perto o projeto de uma solução atingiu seu propósito.

trabalho que, nos testes comparativos entre IRIS e técnicas Postpass convencionais, o tempo de compilação melhorou 38% em média.

17. On Minimizing Register Usage of Linearly Scheduled Algorithms with Uniform Dependencies

Philippidis & Shang [2010] apresentam uma abordagem para minimizar a pressão por registradores em laços aninhados, utilizando-se escalonamento linear (*linear scheduling*) e *loop unrolling*. *Linear scheduling* é uma forma específica de escalonamento de instruções utilizada para a movimentação do código existente dentro de um laço. Os autores relatam que os resultados obtidos indicam que a técnica possui grande potencial e pode ser útil para aplicações de diversas áreas.

18. Register Allocation with Instruction Scheduling for VLIW-architectures

Ivanov [2010] tenta combinar as tarefas de alocação e escalonamento da melhor maneira possível, levando-se em consideração as características do escalonamento estático para arquiteturas VLIW. O termo *web* é utilizado para designar um conjunto de nós e arestas do grafo de fluxo de controle de um procedimento cuja execução requer que valores intermediários de uma variável sejam armazenados. As tarefas de alocação e escalonamento são realizadas utilizando as *webs* identificadas. O autor relata que os testes realizados tiveram bom desempenho da abordagem proposta em relação às estratégias de alocação em códigos já escalonados.

19. Register Allocation with Instruction Scheduling: A New Approach

Pinter [1993, 1996] propôs um grafo denominado *Grafo de Interferência Paralelizável* e provou que a coloração ótima desse grafo implica a alocação dos registradores sem a criação de falsas dependências, ou seja, mantendo-se todas as opções de paralelismo para o escalonador de instruções.

O objetivo de Pinter é encontrar um mapeamento de registradores para o qual o custo total de derramamentos para a memória seja mínimo. Caso o número de registradores seja muito pequeno e haja necessidade de derramamentos para a memória, heurísticas para o escalonamento e para a alocação podem ser aplicadas no grafo simultaneamente.

Na estratégia apresentada, a alocação de registradores é feita em primeiro lugar, utilizando o grafo de interferência paralelizável, e o escalonamento de instruções é feito em seguida. Pinter também apresenta uma generalização do grafo de interferência

paralelizável para cobrir escalonamentos globais, além dos limites dos blocos básicos individuais.

O método apresentado foi implementado em um compilador baseado no GCC, e quatro *benchmarks* do conjunto SPEC92 foram compilados para o processador Intel Pentium. Os experimentos feitos indicaram um ganho médio de 16,75% no desempenho desses *benchmarks*.

20. Minimizing Schedule Length via Cooperative Register Allocation and Loop Scheduling for Embedded Systems

Para minimizar a duração do escalonamento dos laços, Huang et al. [2011] propõem o algoritmo CRRA (*Cooperative Re-scheduling Register Allocation*), no qual o problema de ordenação entre a alocação e o escalonamento é levado em consideração. Inicialmente, realiza-se um escalonamento inicial sem considerar quaisquer questões relacionadas aos registradores. Logo após, durante a fase de alocação dos registradores, a duração geral do escalonamento é avaliada e o próprio alocador poderá fazer novos escalonamentos, manipulando os laços, a fim de eliminar os derramamentos de maior custo e minimizar a duração geral do escalonamento. Como técnica para manipular os laços, utilizou-se *Rotation Scheduling* [Chao & LaPaugh, 1993].

A originalidade dessa proposta está no fato de que é o próprio alocador de registradores o responsável pela solução do problema de ordenação das fases, uma vez que ele é capaz de balancear as fases de escalonamento e alocação. Os resultados obtidos indicam que a técnica proposta pode, em média, reduzir em 12% a duração do escalonamento em relação a outras abordagens, como RCRS (*Register Constrained Rotation Scheduling*) [Wang et al., 1998].

21. Variable Assignment and Instruction Scheduling for Processor with Multi-module Memory

Zhang et al. [2011] criaram PEOS (*Performance and Energy Optimal Scheduling*) para otimizar o desempenho e o consumo de energia em multimódulos de memória (*multi-module memory*). Multimódulos de memória possuem modos de operação de baixo consumo de energia e possibilitam acessos simultâneos à memória, quando as variáveis acessadas estão armazenadas em módulos distintos. Para que se tenha ganho com o uso dessas características, entretanto, o compilador precisa gerar um código otimizado para essas arquiteturas.

PEOS é uma técnica baseada em programação linear inteira (ILP - *Integer Linear Programming*), cujo objetivo é gerar um escalonamento com menor tempo de execução e

mínimo consumo de energia. Para atingir esse objetivo, PEOS utiliza um único modelo ILP e busca a solução simultânea para os três seguintes problemas: ❶ atribuição de variáveis para os módulos de memória considerando o desempenho e o consumo de energia; ❷ escalonamento de instruções; ❸ definição do modo de operação de cada módulo de memória em cada passo de controle. Constrói-se o modelo ILP com base na configuração do multimódulo de memória existente na arquitetura e em uma estrutura de dados própria, o grafo MDFG (*Memory-access Data Flow Graph*), que descreve o programa utilizado.

PEOS foi testada utilizando-se um conjunto de *benchmarks* em um conjunto de processadores DSP (*Digital Signal Processor*). O algoritmo para a transformação dos *benchmark* escritos em C para o grafo MDFG foi escrito pelos autores e a resolução do modelo ILP gerado foi feita por uma ferramenta (*solver*) comercial. Os resultados dos testes foram comparados com os resultados de técnicas que também buscam otimizar o desempenho e o consumo de energia de multimódulos de memória, mas não de forma integrada e simultânea. Segundo Zhang et al., PEOS mostrou ser uma técnica promissora, apresentando melhores resultados em todas as comparações realizadas e mostrando ser capaz de atingir um bom compromisso entre o desempenho e o consumo de energia. Para alguns casos, o tempo gasto para a solução do problema ILP foi considerado aceitável pelos autores, todavia, na maior parte dos casos, esse tempo foi elevado, indicando a necessidade de aperfeiçoamento da técnica.

22. Constraint-Based Register Allocation and Instruction Scheduling

Lozano et al. [2012] tratam a alocação global e o escalonamento local como um problema de otimização, *Constraint Programming*. O problema é modelado considerando-se as instruções do programa, as características da arquitetura alvo e as restrições específicas para a alocação e para o escalonamento. Os autores criaram uma nova forma de representação do código intermediário, denominada LSSA (*Linear SSA*, ou *Linear Static Single Assigment*), e a utilizaram na modelagem do problema. Os resultados obtidos indicam que a qualidade do código gerado é similar a do código produzido pelo LLVM.

23. Register Allocation for Fine Grain Threads on Multicore Processor

Kiran et al. [2015] propõem duas heurísticas para alocação de registradores em arquiteturas *multicore*. A partir da identificação e extração das regiões de paralelismo (*fine grained threads*) [Kiran et al., 2011a], constrói-se o grafo de dependência entre essas regiões e efetua-se o escalonamento para cada núcleo do processador [Kiran et al., 2011b, 2012]. A alocação é realizada aplicando-se a técnica de coloração [Chaitin et al., 1981]

no grafo de dependência criado. As heurísticas propostas mostraram-se mais eficientes do que os métodos desenvolvidos para processadores com um único núcleo (*unicore*).

24. Register Allocation and Instruction Scheduling in Unison

Lozano et al. [2016] propuseram uma ferramenta chamada *Unison*, baseada em modelos combinatórios, para realizar a alocação de registradores e o escalonamento de instruções de forma integrada. Unison utiliza as informações da arquitetura alvo para criar modelos distintos para as tarefas de alocação e escalonamento. Em seguida, integra esses dois modelos, utilizando como referência o tempo de vida das variáveis. Esse modelo integrado é então fornecido a um *solver*, que gera o código objeto. Os testes realizados indicam que, para arquiteturas simples, a qualidade do código gerado por Unison é similar a do código gerado pelo LLVM. Para arquiteturas complexas, a qualidade do código gerado por Unison é superior a do código gerado pelo LLVM.

25. Register Allocation and Promotion Through Combined Instruction Scheduling and Loop Unrolling

Domagala et al. [2016] buscam otimizar a alocação de registradores para os laços (*loops*) de um programa. Os autores implementaram no compilador *Open64* uma abordagem que considera os impactos do escalonamento e da técnica de *loop unrolling* sobre as tarefas de alocação de registradores e promoção de registradores [Lo et al., 1998]. Os testes da abordagem proposta foram realizados na arquitetura x86, considerando configurações com 8 e 16 registradores disponíveis. Utilizando os *benchmarks* da suíte SPECint, 6323 laços internos foram analisados. A abordagem proposta foi comparada com os algoritmos de promoção de registradores e de alocação de registradores existentes no compilador Open64. Pelos gráficos apresentados no trabalho, é possível estimar que, em média, o método proposto reduziu em 10% o número de derramamentos para a memória, sendo que os casos de maior pressão sobre os registradores foram os que mais se beneficiaram da abordagem sugerida.

3.3.2 Discussão dos Resultados

Nesta seção, apresentamos respostas para as questões de pesquisa definidas na revisão sistemática. Tais respostas foram produzidas a partir dos dados coletados dos estudos primários selecionados.

QP1. Quais são as principais abordagens propostas na literatura para lidar com o problema da interdependência entre as tarefas de alocação de

registradores e escalonamento de instruções?

Os estudos primários analisados indicam que duas abordagens têm sido utilizadas para a resolução do problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções: a *abordagem cooperativa* e a *abordagem integrada*.

Na abordagem cooperativa, as tarefas de alocação e escalonamento são executadas separadamente, contudo, cada uma delas, ao ser processada, leva em consideração as necessidades da outra. Geralmente, as tarefas são realizadas em momentos distintos e não trocam informações entre si. Logicamente, apenas a primeira tarefa a ser executada considera as necessidades da próxima. Isso não ocorre apenas quando as tarefas são executadas múltiplas vezes [Brasier et al., 1995], ou quando otimizações locais e globais são realizadas separadamente [Norris & Pollock, 1998]. A tarefa que está sendo processada considera as necessidades da outra por meio de heurísticas, ou critérios inseridos no método que ela utiliza.

Na abordagem integrada, tenta-se resolver o problema da interdependência a partir da realização conjunta das duas tarefas. Isso não quer dizer que dois processos distintos, a alocação de registradores e o escalonamento de instruções, são processados em paralelo. Geralmente, executa-se um método único capaz de realizar o trabalho das tarefas de alocação e escalonamento simultaneamente.

Como exemplo de abordagem integrada, podemos citar a ferramenta Unison [Lozano et al., 2016], que trata as tarefas de alocação e escalonamento como partes de um único problema de otimização combinatória. A solução desse problema implica a realização conjunta das duas tarefas. Para exemplificar a abordagem cooperativa, podemos citar a solução apresentada em Pinter [1996], na qual o *grafo de interferência paralelizável*, utilizado como estrutura de dados pela tarefa de alocação de registradores, é construído levando-se em consideração as necessidades da tarefa de escalonamento.

As figuras 3.4 e 3.5 apresentam dois gráficos com informações sobre as duas abordagens identificadas.

Por meio desses gráficos, é possível verificar que a abordagem mais utilizada é a cooperativa, aparecendo em 56% dos estudos. A abordagem integrada é utilizada em 40% dos estudos. Um dos estudos selecionados, Valluri & Govindarajan [1999], apenas avalia soluções existentes e não busca solucionar o problema com uma abordagem específica. Por esse motivo, ele aparece como *Não aplicável* nos gráficos.

Em Bradlee et al. [1991], ambos os métodos IPS e RASE, que tratam as tarefas de alocação e escalonamento de forma cooperativa e integrada respectivamente, mostraram-se mais eficientes para solucionar o problema de interferência do que o mé-

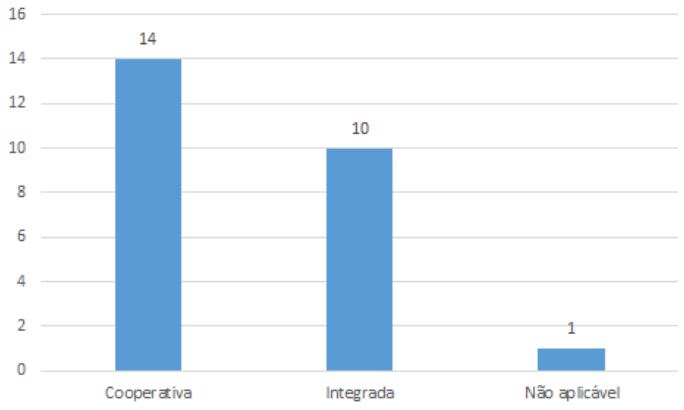


Figura 3.4. Total de estudos por tipo de abordagem.

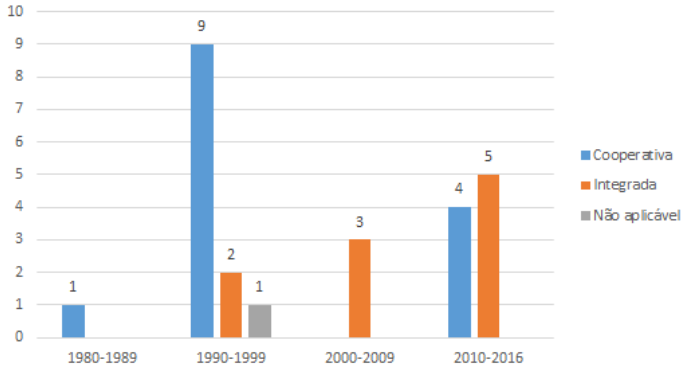


Figura 3.5. Total de tipos de abordagem por década de publicação dos estudos.

todo *postpass* convencional [Hennessy & Gross, 1983; Gibbons & Muchnick, 1986], que trata essas tarefas separadamente. Contudo, o método RASE, que aborda o problema de forma integrada, não trouxe ganhos que justificassem a sua utilização em substituição ao método IPS, que é cooperativo. Os próprios autores indicaram a utilização do método IPS e, por esse motivo, computamos esse estudo para a abordagem cooperativa.

Sumário: os estudos primários analisados sugerem que as principais abordagens para lidar com a interdependência entre a alocação de registradores e o escalonamento de instruções são as abordagens cooperativa e integrada, sendo que a primeira delas foi utilizada em um número maior de estudos.

QP2. A consideração das tarefas de alocação de registradores e escalonamento de instruções *de forma conjunta* realmente melhora a otimização do código?

Pela revisão sistemática realizada, uma *abordagem conjunta*[9] das tarefas melhorou a otimização do código em 80% dos estudos analisados. A Figura 3.6 apresenta o gráfico com os totais para cada uma das opções de resposta dessa questão de pesquisa. Como indica o gráfico, apenas três estudos não apresentaram códigos mais otimizados utilizando uma abordagem conjunta. Além disso, esta questão não foi aplicável a dois estudos.

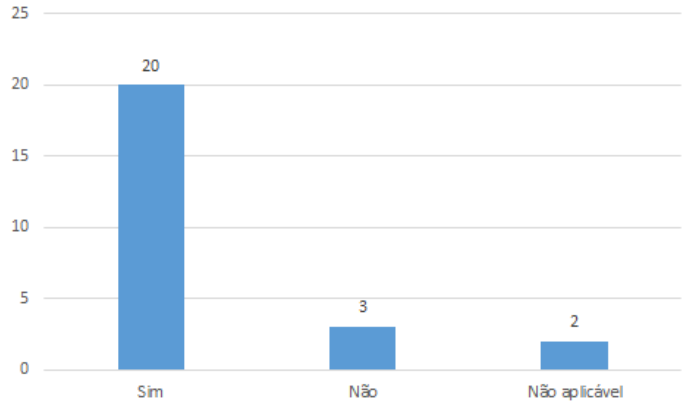


Figura 3.6. Respostas para a Questão de Pesquisa 2.

Os trabalhos de Chang et al. [1997] e Zhang et al. [2011] são dois dos três casos em que uma abordagem conjunta para a solução do problema não melhorou a otimização do código. Esses estudos descrevem abordagens *integradas*, que utilizam programação linear inteira (ILP) para a resolução simultânea das tarefas de alocação e escalonamento. Em ambos, o tempo para a obtenção das soluções foi muito alto, inviabilizando a utilização prática das técnicas propostas.

Em Valluri & Govindarajan [1999], encontramos o terceiro caso em que uma abordagem conjunta não melhorou a otimização do código. Os autores testaram quatro abordagens de solução, propostas por Gibbons & Muchnick [1986],Goodman & Hsu [1988] e Pinter [1993], em um processador com execução de instruções fora de ordem. Dentre essas abordagens, duas são cooperativas e duas são abordagens tradicionais com a execução separada das tarefas de alocação e escalonamento. Nos testes realizados, utilizou-se um simulador e não uma máquina real. De acordo com os autores, para os processadores com execução de instruções fora de ordem, o processo de geração de código deveria apenas evitar a pressão sobre os registradores e os derramamentos de variáveis para a memória, sem se preocupar com a tarefa de escalonamento de instruções. Eles verificaram que as técnicas que consideram a cooperação entre as tarefas de alocação e escalonamento mostraram melhoras insignificantes no desempenho,

[9]Uma abordagem cooperativa ou uma abordagem integrada.

quando comparadas com às técnicas *prepass* e *postpass* sem qualquer tipo de integração. Para *benchmarks* com alta pressão sobre os registradores, as técnicas *postpass* e *postpass-like*[10] avaliadas geraram códigos com melhor desempenho, pois priorizaram a alocação de registradores. Para os autores, esses resultados se devem às características do processador com execução de instruções fora de ordem, que é capaz de escalonar as instruções dinamicamente utilizando mecanismos de hardware.

Os dois estudos para os quais esta questão de pesquisa não foi aplicada são Govindarajan et al. [2000] e Depuydt et al. [1994]. No primeiro, os autores propuseram uma abordagem integrada que realiza as tarefas de alocação de registradores e escalonamento de instruções para processadores com execução de instruções fora de ordem. Porém, os testes da abordagem proposta não foram realizados e foram sugeridos como um trabalho futuro. No segundo estudo, os autores afirmaram a eficiência da técnica cooperativa proposta, realizando experimentos em uma ferramenta para o desenvolvimento de protótipos. Contudo, resultados comparativos entre o método desenvolvido e outros métodos não foram apresentados, de modo que optamos por não avaliar esse estudo nesta questão de pesquisa.

Dentre os tipos de abordagem conjunta, a abordagem cooperativa foi a que produziu o maior número de casos em que se observou melhora na otimização do código. Em 93% dos casos que utilizaram essa abordagem, um código mais eficiente foi produzido, enquanto que, nos casos em que se utilizou a abordagem integrada, uma melhora do código ocorreu em 78% dos casos. As Figuras 3.7 e 3.8 apresentam totalizações das respostas para esta questão de pesquisa. A primeira separa as totalizações pelo tipo de abordagem utilizada. A segunda estratifica essas totalizações pelo tipo de abordagem e também pela década em que a solução que a utiliza foi apresentada.

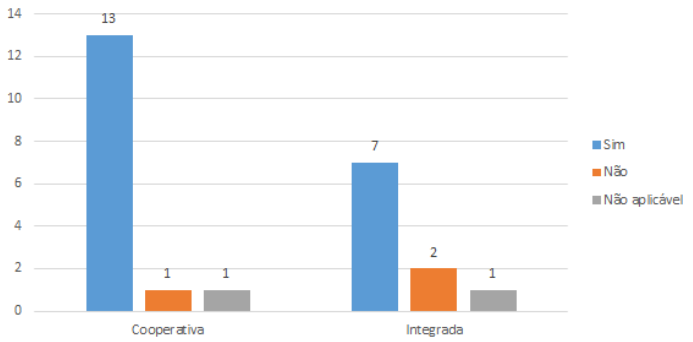


Figura 3.7. Respostas para a QP2 por tipo de abordagem.

[10]Método descrito em Pinter [1993].

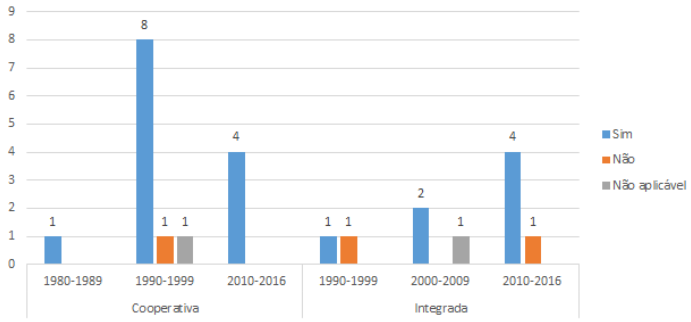


Figura 3.8. Respostas para a QP2 por tipo de abordagem e década em que foi apresentada.

Finalmente, é importante destacar que algumas das abordagens conjuntas consideradas nas estatísticas dessa questão de pesquisa possuem certas restrições. No trabalho de Norris & Pollock [1995], a estratégia cooperativa RASER produz bons resultados somente para arquiteturas com poucos registradores. Para máquinas com um número elevado de registradores, nas quais é mais relevante reordenar o código para aproveitar as oportunidades de paralelismo do que para reduzir o número de derramamentos para a memória, muitas oportunidades de escalonamento são perdidas e o código gerado não apresenta bom desempenho. O algoritmo criado por Kessler [1998] trata o escalonamento de instruções de forma local e trabalha somente com blocos básicos pequenos e médios, de no máximo 50 instruções. Para blocos básicos maiores, o algoritmo não possui bom desempenho e, segundo os autores, não deve ser utilizado.

Sumário: a resposta para esta questão de pesquisa é afirmativa, visto que, na maioria dos estudos primários analisados, a abordagem conjunta das tarefas de alocação de registradores e escalonamento de instrução melhorou a otimização do código. Códigos com melhor desempenho foram gerados tanto em abordagens cooperativas, quanto em abordagens integradas.

QP3. Para quais arquiteturas as tarefas de alocação de registradores e escalonamento de instruções têm sido realizadas *conjuntamente*?

A maior parte dos estudos analisados nesta RSL consideraram arquiteturas que seguem o modelo geral de uma arquitetura RISC, contudo, alguns estudos trabalham com arquiteturas específicas, ou indicam arquiteturas com características particulares.

Em Ivanov [2010], Brasier et al. [1995] e Berson et al. [1994], arquiteturas VLIW (*Very Long Instruction Word*) ou LIW (*Long Instruction Word*) são utilizadas. A diferença entre os termos VLIW e LIW é bastante arbitrária e, geralmente, eles descrevem

o mesmo tipo de arquitetura. Trata-se de uma arquitetura que codifica várias operações em uma única instrução, e cada operação é executada paralelamente em uma das unidades de execução do processador. A garantia de independência entre essas instruções deve ser dada pelo compilador.

Nos trabalhos de Valluri & Govindarajan [1999] e Govindarajan et al. [2000], as abordagens propostas são para processadores com execução de instruções fora de ordem, que são capazes de escalonar as instruções dinamicamente utilizando mecanismos de hardware. Em Kiran et al. [2015], os autores consideram o problema de alocação e escalonamento voltados para processadores *multicore* com suporte a *fine grain parallelism*[11] e realizam testes utilizando processadores *dual-core* e *quad-core*. Em Kim & Lee [2010], os autores não mencionam se a abordagem proposta é voltada para um tipo de processador específico. Porém, na parte de descrição dos testes e avaliação dos resultados, os autores informam que a implementação visou os processadores ARM7TDMI, afirmando que este é o processador de 32-bits mais utilizado em sistemas embarcados no mundo. Em Pinter [1996], o método desenvolvido considera um modelo genérico de uma arquitetura RISC. Os testes foram realizados em um processador *Pentium* com dois *pipelines* não simétricos, que compartilham unidades funcionais. O método apresentado por Depuydt et al. [1994] visou processadores com a arquitetura ASIP, muito utilizados em sistemas embarcados para aplicações DSP de tempo real. Em Zhang et al. [2011], o método desenvolvido destina-se à otimização do desempenho e à redução do consumo de energia de multi-módulos de memória (*multi-module memory*) utilizados em processadores voltados para aplicações DSP. Os testes realizados consideraram um conjunto representativo de processadores DSP e foram feitos em uma máquina com processador *Pentium 4*.

O gráfico da Figura 3.9 apresenta o total de estudos primários avaliados por tipo de arquitetura. As arquiteturas citadas nos estudos que se assemelham à arquitetura RISC, tais como as arquiteturas VLIW, LIW e Superescalar, mas que não foram associadas a um tipo de processador específico, estão agrupadas nesse gráfico sob o termo *Geral*. Pelo gráfico, verifica-se que 76% dos estudos avaliados estão relacionados à arquitetura *Geral*, enquanto 24% deles estão relacionados a arquiteturas específicas.

Sumário: verifica-se que a maior parte dos estudos avaliados, cerca de 76% deles, estão relacionados a uma arquitetura genérica do tipo RISC, enquanto 24% estão relacionados a arquiteturas específicas.

[11] Divisão de um programa em um grande número de tarefas paralelizáveis, cada qual com um pequeno número de instruções.

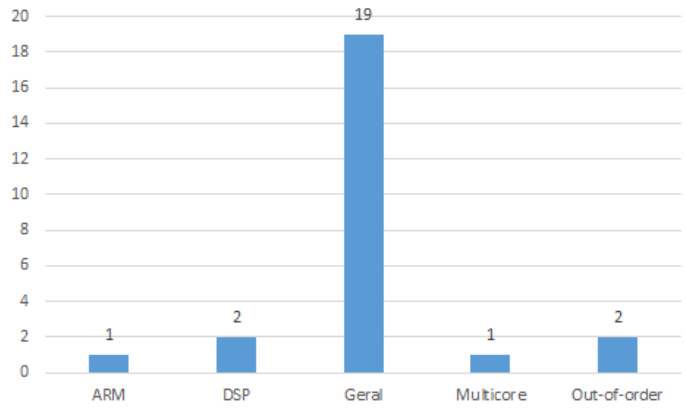


Figura 3.9. Total de estudos por tipo de arquitetura.

3.4 Apresentação

Segundo Kitchenham & Charters [2007], a fase final de uma revisão sistemática compreende a escrita dos resultados encontrados e a divulgação desses resultados para a comunidade de interesse. Devido a isso, os resultados obtidos nessa revisão sistemática também foram registrados e publicados em um artigo apresentado no XXI SBLP (Simpósio Brasileiro de Linguagens de Programação), durante o VIII CBSoft (Congresso Brasileiro de Software).

3.5 Ameaças à Validade

Nesta seção, listamos e discutimos três tipos de ameaça que podem ter comprometido a validade da revisão sistemática realizada: ameaças internas, ameaças de construção e ameaças de conclusão [Cook & Campbell, 1979; Wohlin et al., 2012]. Para cada uma delas, indicamos as ações de mitigação que foram tomadas.

3.5.1 Ameaças Internas

Em uma revisão sistemática de literatura, o *termo de busca* deve ser definido de tal modo que as pesquisas retornem o maior número possível de estudos primários relacionados ao tema pesquisado. Caso o *termo* não seja bem definido, o número de estudos retornados no processo de busca pode ser muito pequeno, ou então estudos relevantes podem não ser encontrados durante as pesquisas. Para mitigar essa ameaça, no planejamento da revisão sistemática, tentamos considerar todas as palavras-chave relacionadas ao tema pesquisado. Também utilizamos vários sinônimos para essas palavras, na tentativa de obter um *termo de busca* de alta qualidade. Além disso, várias

pesquisas experimentais foram realizadas com a intenção de identificarmos as melhores palavras-chave e os melhores sinônimos, antes da definição final do *termo*. Assim, é de se esperar que o *termo de busca* definido seja de boa qualidade e tenha possibilitado o retorno do maior número possível de estudos primários. Todavia, mesmo com as ações tomadas para mitigar esse risco, não é possível afirmar que todos os estudos relacionados ao tema de pesquisa foram encontrados.

A busca em uma base de dados é feita utilizando-se um *termo de busca*. Assim, é possível que uma pesquisa, utilizando um *termo de busca* elaborado especificamente para a base de dados que está sendo pesquisada, produza resultados melhores do que aquela que utiliza um *termo de busca* genérico, criado para pesquisas em quaisquer bases de dados. Na tentativa de solucionar esse problema, várias pesquisas experimentais foram realizadas em todas as bases de dados escolhidas, a fim de identificarmos as melhores palavras e a melhor concatenação lógica entre essas palavras para a composição do *termo de busca*. Em função dessas pesquisas, não foi observada nenhuma vantagem em se utilizar termos distintos. A falta de resultados para a pesquisa feita na base de dados *IEEE Xplore* não pareceu ser um problema, pois nenhum estudo relevante foi retornado, quando se fez a pesquisa nessa base com termos diferentes. Se, por um lado, é possível supor que a utilização de um *termo de busca*, criado especificamente para uma determinada base, possa produzir resultados melhores nessa base, por outro lado, é inegável que a utilização de um termo único simplifica e padroniza o processo de busca. Em função dessa simplificação e dos resultados das pesquisas experimentais, a opção de se utilizar um termo de busca único foi a escolhida.

As bases de dados têm influência direta nos resultados da revisão sistemática, pois elas são as fontes dos estudos primários que serão analisados. Bases não relacionadas ao tema de pesquisa, ou com poucos estudos armazenados, comprometem os resultados da revisão. Para mitigar esse problema, seis bases de dados eletrônicas foram escolhidas como fontes de estudos primários, uma quantidade que acreditamos ser suficiente. Mais do que isso e principalmente, todas elas possuem uma grande quantidade de estudos completos, publicados como artigos ou relatórios técnicos, em conferências e revistas relevantes para a comunidade acadêmica associada ao tema de pesquisa da revisão. É razoável supor que, em função disso, o risco tenha sido bastante reduzido, contudo, estudos relevantes podem existir em bases de dados que não foram consideradas.

3.5.2 Ameaças de Construção

A seleção dos estudos primários para análise é um momento crítico da revisão sistemática, pois há risco de se excluir estudos relevantes para a revisão, comprometendo

o seu resultado. No caso particular da revisão sistemática realizada, as Etapas 2 e 3 do processo de seleção basearam-se na avaliação de metadados dos estudos primários, isto é, basearam-se na avaliação do título e do resumo dos estudos. Assim, estudos relevantes podem ser desconsiderados, caso seus metadados não tenham uma relação clara com o tema de pesquisa, ou caso sejam avaliados incorretamente. Para tratar essa ameaça, a seleção dos estudos primários não foi realizada por uma única pessoa. Três avaliadores participaram do processo de seleção. Todas as exclusões feitas na Etapa 3 foram autorizadas por, no mínimo, dois desses três avaliadores. Além disso, ainda nessa etapa, todos os estudos foram avaliados por todos os avaliadores. Na Etapa 2, no entanto, os estudos foram distribuídos entre os avaliadores e a exclusão de um estudo pôde ser autorizada pelo avaliador que o recebeu.

3.5.3 Ameaças de Conclusão

Ressaltamos dois riscos que foram identificados, mas que não receberam uma ação específica de mitigação.

O primeiro deles está relacionado ao *critério de exclusão de teses e dissertações*. Importantes avanços em várias áreas da ciência têm surgido a partir de trabalhos acadêmicos como teses de doutorado. Geralmente, dissertações de mestrado têm sido menos inovadoras, mas também há exceções a isso. Dessa forma, o critério que definiu a exclusão de teses e dissertações gera o risco de que avanços relevantes sobre o tema de pesquisa sejam desconsiderados. Mesmo assim, optamos por manter esse critério, devido a um fato bastante comum, que serve para mitigar o risco existente. É bastante usual que artigos científicos sejam produzidos com base em teses e dissertações, ainda mais quando esses trabalhos trazem inovações. Portanto, é plausível supor que tais trabalhos acadêmicos possam ser excluídos durante o processo de seleção dos estudos, pois artigos científicos correspondentes serão encontrados e, estando de acordo com os critérios de inclusão, serão considerados.

O segundo está relacionado ao critério de inclusão *estudos escritos em inglês*. O tema de pesquisa considerado é estudado em diversos países e há risco de que estudos relevantes sobre ele tenham sido desconsiderados, em função desse critério de inclusão. Todavia, é sabido que, já há algum tempo, todos os estudos de ponta sobre quaisquer temas de pesquisa têm sido publicados em inglês. Não há regra que determine isso, mas essa prática acabou se tornando um padrão no meio acadêmico e científico. Dessa forma, o risco mencionado foi desconsiderado, por ter sido considerado muito pequeno, e esse critério de inclusão foi mantido.

3.6 Considerações Finais

Neste capítulo, apresentamos a revisão sistemática da literatura sobre o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Essa interdependência é um tema relevante na área de compiladores e representa um problema de difícil solução, para o qual uma solução ótima não pode ser encontrada em tempo polinomial. Com a revisão sistemática realizada, uma visão abrangente sobre o assunto pôde ser desenvolvida e as abordagens propostas na literatura para lidar com o problema puderam ser identificadas.

O objetivo do alocador de registradores é minimizar o número de acessos à memória em tempo de execução, mas, geralmente, ele interfere no escalonamento de instruções, cujo objetivo é reordenar as instruções para maximizar as possibilidades de paralelismo. O problema é que ainda não se sabe qual deles deve ser executado em primeiro lugar, nem como executá-los de modo que ambos alcancem a melhor solução possível. Se o alocador for o primeiro a ser executado, variáveis de instruções independentes podem ser atribuídas a um mesmo registrador, limitando as possibilidades de escalonamento. Se, ao contrário, o escalonador for o primeiro, os registradores podem ficar ocupados por muito tempo, impondo um maior número de acessos à memória.

Vimos que, tradicionalmente, duas abordagens principais têm sido utilizadas para lidar com o problema: a abordagem cooperativa e a abordagem integrada. Pela revisão sistemática realizada, a abordagem cooperativa foi utilizada em 56% dos estudos analisados, enquanto a abordagem integrada foi utilizada em 40% desses estudos. Essas abordagens produziram códigos com melhor desempenho em 80% dos estudos analisados, sendo que a abordagem cooperativa melhorou a otimização do código em 93% dos casos e a abordagem integrada em 78% dos casos. Considerando as arquiteturas utilizadas, cerca de 76% dos estudos utilizaram modelos genéricos de uma arquitetura RISC, enquanto arquiteturas específicas foram utilizadas em 24% desses estudos.

É interessante destacar que apesar de se tratar de um problema antigo, que já vem sendo considerado há tempos, não há dúvidas de que ainda se trata de um problema que está aberto a novas soluções e avaliações. Estudos sobre o tema comprovam isso e soluções utilizando abordagens inovadoras têm sido propostas, como podemos ver em, por exemplo, Kri & Feeley [2004], Lozano et al. [2012] e Lozano et al. [2016].

Capítulo 4

Uma Solução para a Interdependência entre AR e EI

Primeiramente, é importante esclarecer o título deste capítulo. Com efeito, ainda não há *uma solução* que resolva de forma cabal e definitiva o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Como mostrou a revisão sistemática sobre o tema, o que há são diferentes abordagens que utilizam técnicas ou heurísticas capazes de gerar resultados satisfatórios para o problema sob determinadas condições.

Este capítulo apresenta e detalha a abordagem escolhida após a análise de todas as que foram identificadas na revisão sistemática da literatura. Tal escolha foi feita com base em uma avaliação qualitativa que considerou os seguintes critérios: *eficiência*, *abrangência* e *flexibilidade*.

A seção 4.1 apresenta a abordagem escolhida e a Seção 4.2 analisa essa abordagem considerando ❶ os critérios de escolha utilizados, ❷ as informações fornecidas no artigo em que ela foi apresentada e ❸ as principais arquiteturas de computador descritas na Seção 2.3.1.

4.1 A Abordagem de Pinter

Em 1993, Shlomit Sarah Pinter propõe a utilização do *Grafo de Interferência Paralelizável* como estrutura de dados para a tarefa de alocação de registradores. Pinter prova que a coloração ótima desse grafo implica a alocação ótima dos registradores sem a criação de falsas dependências. Sendo assim, a alocação de registradores feita com a utilização do grafo de interferência paralelizável mantém todas as opções de paralelismo para o escalonador de instruções [Pinter, 1993, 1996].

A motivação para a definição do grafo de interferência paralelizável vem do fato de que, se consideradas isoladamente, as tarefas de alocação de registradores e escalonamento de instruções utilizam estruturas de dados distintas e inconciliáveis. A tarefa de alocação utiliza o grafo de interferência, no qual os vértices representam variáveis temporárias do código intermediário. A tarefa de escalonamento, por sua vez, utiliza o grafo de escalonamento, no qual os vértices representam as instruções desse código.

Para um único bloco básico, o grafo de interferência paralelizável deve ser construído de acordo com os passos listados a seguir e ilustrados na Figura 4.1:

1. criar o *grafo de escalonamento* a partir do código intermediário;
2. fazer o *fechamento transitivo* do grafo de escalonamento;
3. retirar as direções das aretas do grafo resultante, transformando o DAG em um grafo não dirigido;
4. inserir no grafo resultante as arestas relativas às restrições da arquitetura alvo;
5. fazer o *complemento* do grafo resultante, gerando o *grafo de falsas dependências*;
6. criar o *grafo de interferência* a partir do código intermediário;
7. incluir as arestas do grafo de falsas dependências no grafo de interferência, gerando o *grafo de interferência paralelizável.*

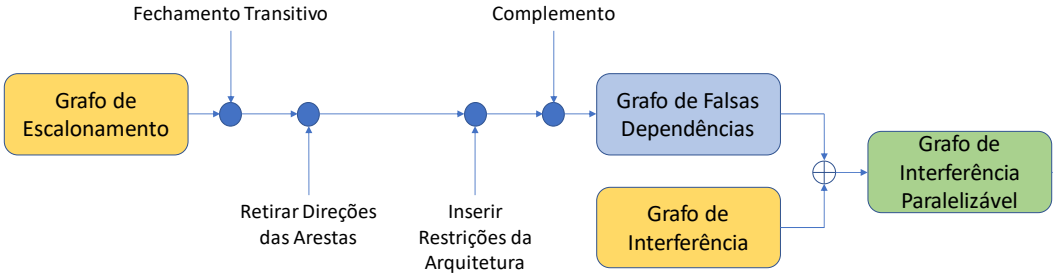


Figura 4.1. Construção do Grafo de Interferência Paralelizável.

Para a realização do *Passo 2*, segundo Cormen et al. [2009], o *fechamento transitivo* de um grafo dirigido $G = (V, E)$, no qual V é o conjunto de vértices e E é o conjunto de arestas, é o grafo dirigido $F = (V', E')$, tal que:

$$V' = V \text{ } e$$
$$E' = \{(u, v) \mid u, v \in V \text{ } e \text{ } há \text{ } um \text{ } caminho \text{ } entre \text{ } u \text{ } e \text{ } v \text{ } em \text{ } G\}$$

Para a execução do *Passo 5*, ainda segundo Cormen et al. [2009], o *complemento* de um grafo $G = (V, E)$, no qual V é o conjunto de vértices e E é o conjunto de arestas,

é o grafo $C = (V', E')$, tal que:

$$V' = V \ e$$
$$E' = \{(u, v) \mid u, v \in V, \ u \neq v \ e \ (u, v) \notin E\}$$

Finalmente, para a realização do *Passo 7*, dado o grafo de falsas dependências $G_f = (V_f, E_f)$ e o grafo de interferência $G_r = (V_r, E_r)$, Pinter define o grafo de interferência paralelizável para um bloco básico como o grafo $G = (V, E)$, tal que:

$$V = V_r \ e$$
$$E = E_r \cup \{(u, v) \mid (u, v) \in E_f \text{ e } u, v \in V\}$$

Para exemplificar o processo de construção do grafo de interferência paralelizável para um bloco básico, o código fonte da Figura 4.2 (a) será utilizado. Esse código gera um único bloco básico representado pelo código intermediário da Figura 4.2 (b).

```
x := a[i]          s1 := load z
y := z + z         s2 := i
z := x * 5 + z     s3 := a[s2]
                   s4 := s1 + s1
                   s5 := s3 * 5 + s1
```

(a) (b)

Figura 4.2. Códigos para a construção do grafo de interferência paralelizável: (a) código fonte; (b) bloco básico.

A partir desse código intermediário, o grafo de interferência paralelizável pode ser obtido seguindo-se os passos indicados. Os resultados de cada um desses passos são os grafos apresentados na Figura 4.3. A Figura 4.3 (a) apresenta o grafo que resulta do *Passo 1*. A Figura (b) apresenta o grafo que resulta do *Passo 2* e assim por diante.

O exemplo apresentado nas figuras 4.2 e 4.3 foi extraído de Pinter [1996]. Nesse artigo, Pinter considerou como modelo uma máquina superescalar com uma única unidade de acesso à memória e uma única unidade aritmética. Por esse motivo, as arestas $(s1, s3)$ e $(s4, s5)$ estão presentes no grafo da Figura 4.3 (d).

Na abordagem de Pinter, após a construção do grafo de interferência paralelizável, as tarefas de alocação de registradores e escalonamento de instruções podem ser realizadas. A primeira a ser executada é a tarefa de alocação, que, utilizando o grafo de interferência paralelizável, busca um mapeamento de registradores que minimize o custo total de derramamentos para a memória. Caso o número de registradores da máquina não seja grande o suficiente para acomodar todas as variáveis vivas no código

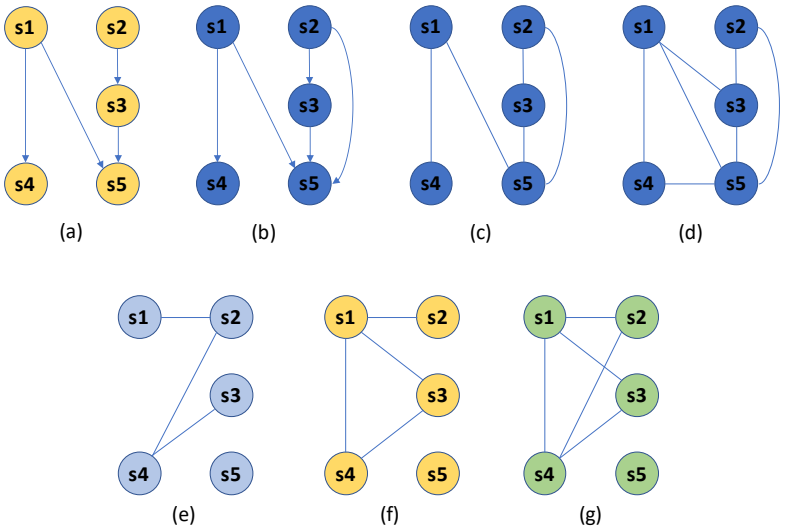


Figura 4.3. Grafos para a construção do grafo de interferência paralelizável: (a) grafo de escalonamento; (b) grafo com fechamento transitivo; (c) grafo não direcionado; (d) grafo com as restrições de máquina; (e) grafo de falsas dependências; (f) grafo de interferência; (g) grafo de interferência paralelizável.

em um determinado momento, sendo necessário o derramamento de valores para a memória, heurísticas relacionadas à alocação a ao escalonamento podem ser utilizadas, a fim de que os valores a serem derramados sejam escolhidos de forma adequada. Finalizada a tarefa de alocação dos registradores, o escalonamento das instruções pode ser efetuado.

Pinter não define em sua abordagem quais métodos devem ser utilizados pelas tarefas de alocação de registradores e escalonamento de instruções. A abordagem proposta não depende de um algoritmo específico para qualquer uma dessas duas tarefas e, em princípio, quaisquer métodos podem ser utilizados por elas. O ponto central da abordagem é a construção do *grafo de interferência paralelizável* e a única exigência é que esse grafo seja considerado no processo de alocação dos registradores.

O processo de criação do grafo de interferência paralelizável pode ser generalizado para permitir a realização de otimizações globais, isto é, para que as tarefas de alocação e escalonamento considerem regiões do código que extrapolam os limites dos blocos básicos. Para isso, Pinter utiliza informações do Grafo de Fluxo de Controle (CFG) do programa e constrói o grafo de interferência paralelizável global executando os mesmos passos ilustrados na Figura 4.1, mas considerando as seguintes observações:

- No *Passo 1*, o *grafo de escalonamento* utilizado deve ser o grafo global, construído levando-se em conta as informações do grafo de fluxo de controle associado à região do código que será otimizada.

- No *Passo 2*, antes do fechamento transitivo do grafo de escalonamento ser feito, todas as arestas que se referem ao fluxo de controle do programa devem ser removidas.
- No *Passo 4*, juntamente com as arestas associadas às restrições de máquina, também devem ser inseridas no grafo arestas entre instruções pertencentes a cada par de blocos básicos que não podem ser escalonados simultaneamente.
- No *Passo 6*, o *grafo de interferência* deve ser o grafo global.

Importante destacar que as três primeiras observações alteram, fundamentalmente, o processo de criação do *grafo de falsas dependências*. A última das observações é facilmente derivada do contexto da situação e é citada por mera formalidade.

Para identificar os blocos básicos que podem ser escalonados simultaneamente, Pinter indica a utilização de análises de dominância e pós-dominância. Essas análises permitem a movimentação de instruções entre os blocos básicos do programa e também possibilitam a identificação dos blocos que podem ser processados de forma paralela. Em resumo, dois blocos básicos que são *equivalentes por controle* são candidatos a serem processados paralelamente. Além disso, a movimentação de instruções entre eles é mais fácil e econômica de ser feita [Pinter, 1996; Aho et al., 2006].

Sendo assim, tomando o grafo de fluxo de controle da Figura 4.4 como exemplo, apenas os blocos básicos *B1* e *B4* são candidatos a serem escalonados simultaneamente. Realmente, *B1* domina e *B4* pós-domina todos os blocos desse grafo. Ao contrário, os blocos *B2* e *B3* não guardam qualquer relação de dominância entre si e, de fato, suas execuções são mutuamente exclusivas[1].

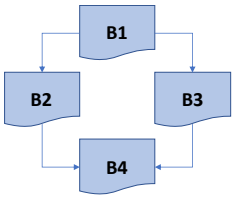


Figura 4.4. Grafo de Fluxo de Controle.

Dessa maneira, conforme as observações feitas e os passos de construção indicados, pode-se concluir que, no grafo de interferência paralelizável associado ao grafo de fluxo

[1]Técnicas de execução especulativa, capazes de processar os blocos *B2* e *B3* simultaneamente, podem ser utilizadas para tentar acelerar a execução do programa. Todavia, tais técnicas são utilizadas em tempo de execução e não alteram a relação de dominância existente entre esses blocos, isto é, não modificam o fato deles serem mutuamente exclusivos. Por essa razão, a desconsideração dessas técnicas não prejudica a análise feita por Pinter.

da Figura 4.4, as instruções de cada par de blocos desse CFG estarão conectadas entre si, exceto para o par de blocos *B2* e *B3*.

4.2 Análise da Abordagem

A estratégia definida por Pinter é um tipo de abordagem *postpass*, uma vez que a alocação de registradores é feita em primeiro lugar, utilizando-se o grafo de interferência paralelizável, e o escalonamento de instruções é feito em seguida. Trata-se de uma abordagem *cooperativa*, na qual a tarefa de alocação leva em consideração aspectos importantes para a tarefa de escalonamento.

Essencialmente, o grafo de interferência paralelizável é o grafo de interferência acrescido das arestas do grafo de falsas dependências, que também foi definido por Pinter. São essas arestas que evitarão o surgimento de falsas dependências no código, quando a alocação dos registradores for feita, pois elas obrigarão a atribuição de registradores distintos para os pseudoregistradores utilizados em instruções que podem ser executadas em paralelo.

A abordagem privilegia o escalonamento de instruções. Isso porque a inserção de arestas no grafo de interferência evita a inclusão de falsas dependências no código, mas pode provocar o derramamento de valores para a memória, uma vez que mais registradores precisarão ser utilizados no processo de alocação. Esse derramamento poderia não ocorrer, caso a alocação de registradores fosse feita sem a inserção de arestas adicionais no grafo de interferência.

Pinter utiliza em seu trabalho uma máquina superescalar como modelo. Essas máquinas baseiam-se fortemente na utilização da técnica de *pipelining* para aumentar o desempenho dos programas. Essa escolha de Pinter não apenas demonstra o foco do seu trabalho, mas também indica o seu interesse de aproveitar ao máximo a capacidade dessa técnica, pois, para isso, privilegiar o escalonamento de instruções é a melhor opção.

Muito importante ressaltar que o modelo superescalar utilizado não possui a capacidade de emitir instruções fora de ordem. De fato, Pinter utiliza em seus testes um processador Intel Pentium, que possui dois *pipelines* de cinco estágios chamados U e V. O primeiro deles é o principal e pode executar qualquer tipo de instrução. O segundo, o *pipeline* V, executa somente instruções aritméticas simples, sejam elas inteiras ou de ponto flutuante. Um conjunto de regras limita a utilização simultânea desses dois *pipelines* e classifica cada uma das instruções a ser processada, determinando assim o *pipeline* que será utilizado para a execução de cada uma delas [Tanenbaum, 2005;

Pinter, 1996].

Apesar da arquitetura superescalar ter sido escolhida como modelo, a abordagem de Pinter não se limita a essas arquiteturas. Na realidade, a abordagem proposta também pode ser utilizada em arquiteturas RISC e até mesmo em arquiteturas CISC, preferencialmente, se elas utilizam a tecnologia superescalar com a iniciação em ordem de instruções.

Os motivos que levaram à escolha da abordagem de Pinter como solução para o problema da interdependência entre as tarefas de alocação de regitradores e escalonamento de instruções foram os seguintes:

- *Eficiência* - Trata-se de uma abordagem que apresentou bons resultados nos testes realizados. Segundo os dados apresentados em Pinter [1996], o desempenho dos *benchmarks* testados melhorou entre 5% e 33%. Em média, houve um aumento de 16,75%, e, em nenhum caso, o desempenho dos *benchmarks* compilados com a abordagem proposta foi inferior, ou mesmo igual ao desempenho dos *benchmarks* compilados com os métodos tradicionais de alocação e escalonamento disponíveis no compilador utilizado, descrito na Seção 3.3.1, Item 19.
- *Abrangência* - Ao contrário de algumas outras abordagens pesquisadas e apresentadas na Seção 3.3.1, que visavam arquiteturas específicas, ou que dependiam de características particulares da máquina, a abordagem de Pinter é bastante genérica e pode ser utilizada para gerar código para todas as principais arquiteturas de computador apresentadas na Seção 2.3.1. Por esse motivo, trata-se de uma das mais abrangentes abordagens encontradas durante o processo de revisão da literatura.
- *Flexibilidade* - A abordagem proposta baseia-se exclusivamente na utilização de uma única estrutura de dados, o grafo de interferência paralelizável. Ela não define técnicas específicas para a realização das tarefas de alocação e escalonamento e isso a torna facilmente adaptável a diversos esquemas de geração de código, principalmente àqueles que já realizam a tarefa de alocação de registradores manipulando algum tipo de grafo. Além disso, métodos mais eficientes de manipulação de grafos que porventura sejam elaborados poderão ser integrados à abordagem com pequeno esforço.

Outro ponto interessante de ser mencionado e que também foi observado para a escolha da abordagem de Pinter, diz respeito à *simplicidade e clareza*. Com efeito, trata-se de uma abordagem extremamente simples e elegante, que pode ser implementada com razoável facilidade em qualquer compilador, uma vez que depende, fun-

damentalmente, de um único grafo, conceito matemático extremamente conhecido e pesquisado, que pode ser implementado e manipulado por diferentes estruturas e algoritmos bastante consolidados.

4.3 Considerações Finais

Neste capítulo, apresentamos em detalhes a abordagem de Pinter, escolhida como solução para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Os principais conceitos e estruturas de dados necessários à utilização dessa abordagem foram explicitados, assim como as razões que motivaram a sua escolha como solução para o problema da interdependência.

No Capítulo 5 apresentamos o ambiente de desenvolvimento escolhido para a implementação da abordagem de Pinter, o LLVM. Apresentamos também o conjunto de *benchmarks* utilizados nos testes da implementação feita, o SPEC CPU2006. Enfatizamos as características mais relevantes dessas ferramentas para a realização do desenvolvimento e dos testes da implementação proposta.

Capítulo 5

Ambiente de Desenvolvimento e Ferramenta de Teste

Este capítulo apresenta o LLVM, o compilador escolhido para a implementação da abordagem de Pinter, evidenciando apenas os conceitos e as características mais relevantes para o desenvolvimento da implementação proposta. Descreve também o conjunto de *benchmarks* SPEC CPU2006 utilizado para os testes da implementação feita.

5.1 LLVM

Quando foi criado na Universidade de Illinois, como um projeto de pesquisa, e descrito em Lattner & Adve [2003] pela primeira vez, LLVM era o acrônimo para a expressão *Low Level Virtual Machine*. Hoje em dia, LLVM não é mais um acrônimo, mas o nome de um portfólio de projetos que formam uma infraestrutura voltada para o desenvolvimento de compiladores [Lattner & Adve, 2004; Team, 2017]. A Figura 5.1 apresenta os projetos e as ferramentas dessa infraestrutura.

Para os objetivos da implementação realizada, os principais projetos desse portfólio são *LLVM Core* e *CLang*. O primeiro deles representa um conjunto de bibliotecas e ferramentas que fornecem várias otimizações de programa independentes de máquina e geram código objeto para diferentes arquiteturas. Novas otimizações e arquiteturas podem ser inseridas no LLVM utilizando-se essas bibliotecas e ferramentas. De modo geral, esse projeto constitui o *backend* de um compilador destinado a múltiplas arquiteturas (*retargetable compiler*) e, portanto, todos os seus componentes utilizam uma representação intermediária do programa fonte, conhecida como *LLVM IR*. Essa representação é construída pelo segundo principal projeto do portfólio, o projeto *CLang*. Com efeito, o objetivo do projeto *CLang* é criar um *frontend* capaz de traduzir, para a

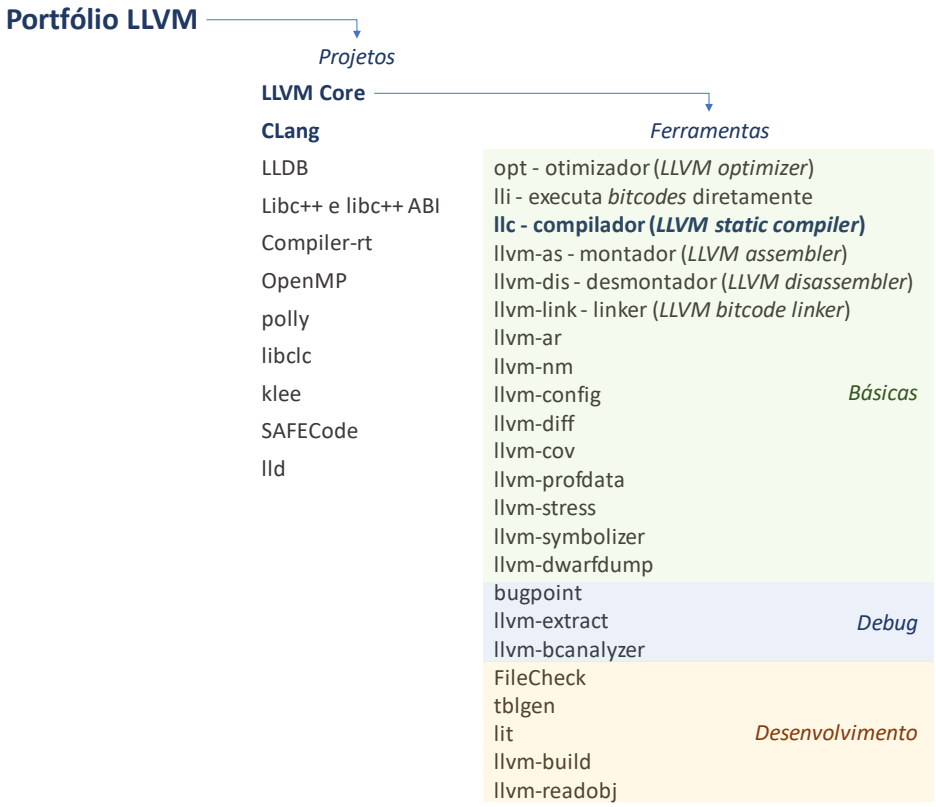


Figura 5.1. Projetos e ferramentas do LLVM.

linguagem intermediária do LLVM, programas fontes escritos em C, C++, Objective-C, Objective-C++, OpenCL C e outras linguagens de alto nível. *CLang* também fornece ferramentas de análise de código, possibilita a tradução de fonte para fonte e suporta todos os padrões C++, desde C++98 até o mais recente C++17.

Com relação às ferramentas existentes no projeto *LLVM Core*, considerando o trabalho descrito nesta dissertação, a mais relevante delas é o compilador *LLC* (*LLVM Static Compiler*). Esse compilador gera códigos objeto a partir de *bitcodes* que foram criados pelo *CLang*. Com o *LLC*, cada um dos passos do gerador de código podem ser configurados por meio de parâmetros utilizados na chamada do compilador. É possível, por exemplo, escolher o método de alocação de registradores desejado, ou estabelecer o momento de execução da tarefa de escalonamento de instruções. A Figura 5.2 ilustra o processo de geração de código do LLVM utilizando o *CLang* e o compilador *LLC*. Também é possível criar o código objeto executando-se apenas *CLang*, que, nesse caso, ficará responsável por chamar internamente as ferramentas necessárias à compilação. Todavia, nesse caso, nem todos os atributos do gerador de código poderão ser escolhidos.

Com relação às arquiteturas, o LLVM pode gerar código para várias delas, al-

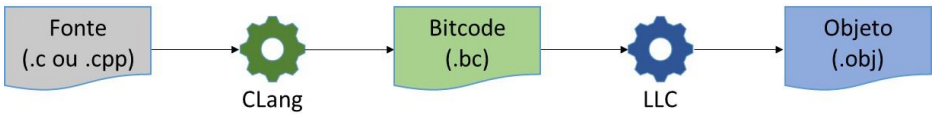


Figura 5.2. Geração do código objeto no LLVM.

gumas mais, outras menos usuais, tais como ARM, Hexagon, Mips, PowerPC, Sparc, SystemZ e X86. Atualmente, o LLVM é capaz de gerar códigos de qualidade a partir de programas C/C++ e Objective-C/C++ para as arquiteturas X86 de 32 e 64 bits, e também para a arquitetura ARM.

Todos os componentes do LLVM estão escritos na linguagem C++. Sendo assim, novos projetos devem, preferencialmente, utilizar essa mesma linguagem. Com relação à documentação, além dos textos disponíveis em Team [2017], as únicas referências encontradas na literatura foram Lopes & Auler [2014]; Sarda & Pandey [2015] e Pandey & Sarda [2015].

Para a implementação da abordagem de Pinter, descrita no Capítulo 4, o gerador de código é o componente mais relevante do LLVM, uma vez que as tarefas de alocação de registradores e escalonamento de instruções são passos realizados por ele. Por essa razão, apresentamos, na próxima seção, informações mais detalhadas sobre o processo de geração de código no LLVM.

5.1.1 Geração de Código no LLVM

O gerador de código traduz o código intermediário fornecido pelo *frontend* do LLVM para um código objeto de uma arquitetura alvo específica. Ele é uma estrutura formada por vários componentes e utiliza uma camada de abstração capaz de representar quaisquer arquiteturas para as quais um código objeto pode ser gerado. No contexto deste trabalho, os componentes mais relevantes dessa estrutura são:

- Camada abstrata de descrição de arquitetura
- Classes de código de máquina
- Algoritmos para a Geração de Código

Além desses três componentes, apenas dois outros existem de fato: a camada MC (*Machine Code*) e a camada JIT (*Just In Time*). O primeiro deles é um conjunto de classes e algoritmos que representam e manipulam as estruturas assembly utilizadas no código objeto, tais como rótulos, seções e instruções. Esse é um componente importante para o processo de geração de código, mas que está associado de maneira mais próxima

aos passos finais do gerador. O segundo desses componentes, a camada JIT, refere-se exclusivamente à compilação dinâmica do código. Esses dois componentes não possuem relação com o desenvolvimento feito.

5.1.1.1 Camada Abstrata de Descrição de Arquitetura

Essa camada funciona como uma interface capaz de representar quaisquer arquiteturas existentes no LLVM. Ela não incorpora partes específicas dos algoritmos de geração de código e não está vinculada a nenhuma arquitetura específica. Trata-se de uma camada de abstração que contribui com a reutilização dos demais componentes do gerador, pois permite que eles sejam independentes dos detalhes específicos de implementação das arquiteturas.

Ela é formada por *classes abstratas de descrição*, que precisam ser herdadas pelas classes de cada uma das arquiteturas implementadas no LLVM. Assim, sempre que uma nova arquitetura é inserida no LLVM, classes derivadas, criadas especificamente para essa nova arquitetura, devem implementar os métodos virtuais das classes de descrição. A Figura 5.3 ilustra a camada abstrata de descrição de arquitetura.

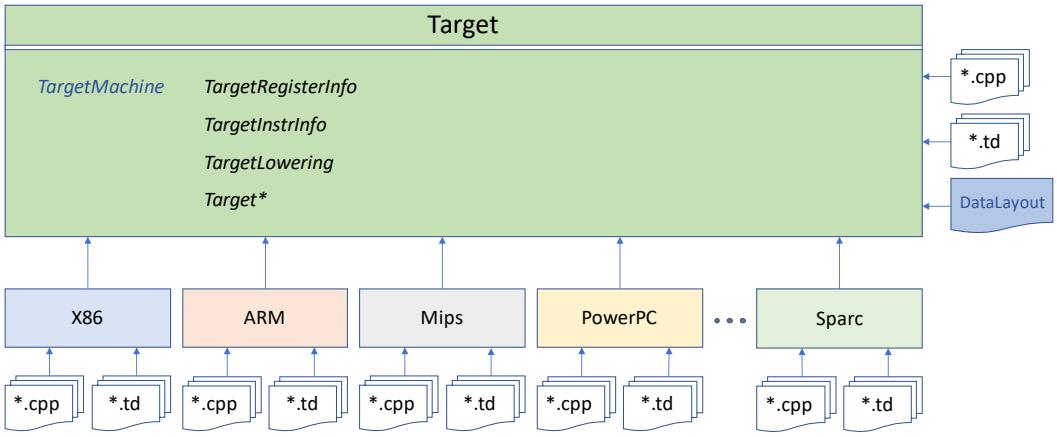


Figura 5.3. Camada de descrição de arquitetura do LLVM.

Para cada uma das arquiteturas, uma descrição detalhada das suas características deve ser fornecida. É preciso informar, por exemplo, quais são as instruções disponíveis, quais são os registradores, os modos de acesso à memória e assim por diante. Essas informações são armazenadas e ficam disponíveis nas próprias classes, mas também em *arquivos de descrição* específicos, que possuem a extensão *.td*.

Nem todas as classes abstratas de descrição precisam ser herdadas para que uma arquitetura seja inserida no LLVM. De fato, a única cujos métodos virtuais precisam ser necessariamente implementados é a classe *TargetMachine*, uma vez que ela é utilizada

para acessar as demais classes de descrição, que podem, ou não existir para uma determinada arquitetura. Além dela, apenas a classe *DataLayout* é obrigatória. Essa é a única classe da camada que não é abstrata e pode ser vista como uma classe final, uma vez que não deve ser herdada por nenhuma outra. Para cada arquitetura, ela define propriedades relacionadas à organização dos dados, tais como critérios de alinhamento na memória, requisitos para a alocação de memória, tamanhos de ponteiros e modos de ordenação (*endianness*).

5.1.1.2 Classes de Código de Máquina

Essas classes armazenam o código intermediário criado pelo *frontend* do LLVM e permitem que ele seja manipulado durante os diversos passos do *backend*, para que seja transformado no código objeto. Elas fornecem uma representação de alto nível, totalmente independente da arquitetura alvo, do código objeto que está sendo gerado.

As classes de código de máquina são as seguintes: *MachineFunction*, *MachineBasicBlock* e *MachineInstr*. O diagrama da Figura 5.4 indica o relacionamento entre elas. Inúmeras outras classes utilizam as classes de código de máquina e associam-se a elas de diversas maneiras. Contudo, todas elas exercem papel auxiliar e podem, ou não ser úteis para o processo de geração de código, dependendo da situação.

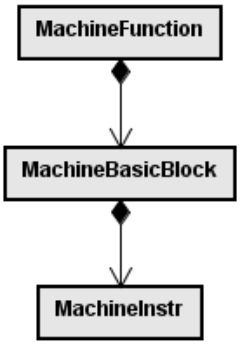


Figura 5.4. Classes que representam o código de máquina no LLVM.

Para o gerador de código, cada instrução do programa é uma instância da classe *MachineInstr*. As instruções armazenadas podem estar na forma SSA (*Static Single Assignment*), como ocorre antes da alocação dos registradores ser feita, ou na forma não-SSA, que é a única possível após a alocação dos registradores.

A classe *MachineBasicBlock* contém uma lista de instruções, ou seja, uma lista de instâncias da classe MachineInstr. Um objeto da classe MachineBasicBlock corresponde a um único bloco básico do código intermediário, todavia um mesmo bloco básico pode

ser mapeado para mais de uma instância dessa classe, para a realização de testes e otimizações por exemplo.

Finalmente, a classe *MachineFunction* contém uma lista de blocos básicos, isto é, uma lista de instâncias da classe MachineBasicBlock. Cada uma das funções do código intermediário é mapeada para uma única instância da classe MachineFunction.

5.1.1.3 Algoritmos para a Geração de Código

Esses algoritmos implementam as várias tarefas necessárias à geração do código objeto, tais como a alocação de registradores e o escalonamento de instruções. Eles manipulam as classes de código de máquina e, gradualmente, transformam a representação independente da arquitetura alvo que existe nessas classes no código objeto destinado a uma arquitetura específica.

O modelo padrão de geração de código do LLVM é formado por sete fases, que trabalham sobre cada uma das funções definidas no programa fonte. Algumas delas são realizadas em um único passo do compilador, outras, como as fases de otimização, podem demandar mais de um passo. Todas elas, conforme seus objetivos, podem utilizar uma ou mais análises do código, tais como análise de variáveis vivas, análise de laços, análise de dominância e assim por diante. Essas análises são construídas de maneira totalmente independente das tarefas realizadas em cada uma dessas fases e podem ser utilizadas por essas tarefas por meio de funções e estruturas específicas.

As fases do modelo de geração de código estão listadas a seguir. Dentre elas estão as tarefas de escalonamento de instruções, *Fase 2*, e alocação de registradores, *Fase 4*. Dada a relevância dessas duas fases para este trabalho, elas serão apresentadas com mais detalhes logo após a descrição das sete fases.

1. **Seleção de instruções** - Nesta fase, o código intermediário é traduzido para um conjunto de instruções da arquitetura alvo, gerando um código objeto inicial, cujas instruções estão na forma SSA e utilizam registradores virtuais. Algumas instruções, em função de restrições de máquina, ou convenções de chamada definidas na arquitetura, podem utilizar registradores diretamente. O método usado para a realização dessa tradução baseia-se em um grafo dirigido, chamado *SelectionDAG*, e em um seletor de instruções. Terminada esta fase, todas as instruções do código estarão armazenadas no grafo.

2. **Escalonamento de instruções** - As instruções do SelectionDAG são ordenadas de acordo com o método de escalonamento definido.

3. **Otimizações do código de máquina** - Trata-se de uma fase opcional, que realiza uma série de otimizações de código baseadas na representação SSA das instruções. *Modulo scheduling* e a técnica *peephole* são exemplos dessas possíveis otimizações.
4. **Alocação de registradores** - Todas as instruções do código passam a utilizar referências para os registradores da máquina alvo, ou para a memória.
5. **Inserção de código inicial e final** - Uma vez que o código objeto para a função foi gerado e a quantidade de espaço na pilha foi determinada, os códigos de início e fim da função são inseridos, e as referências abstratas para a pilha são eliminadas.
6. **Otimizações finais** - Otimizações que podem ser realizadas no código objeto final, tal como a otimização *peephole*, são realizadas nesta fase.
7. **Emissão de código** - Nesta última fase, o código objeto gerado é emitido como código *assembly* para a máquina alvo.

Escalonamento de Instruções

A tarefa de escalonamento de instruções utiliza como estrutura básica o grafo dirigido *SelectionDAG*. Concluída a execução dessa tarefa, cada uma das instruções do código estará representada por uma instância da classe *MachineInstr* e o DAG utilizado será destruído.

No modelo padrão de geração de código do LLVM, o escalonamento de instruções é realizado antes da alocação de registradores. A abordagem *prepass* é portanto a abordagem padrão. Contudo, por meio de parâmetros passados ao compilador, é possível alterar esse comportamento, desabilitando o escalonamento *prepass* e habilitando o escalonamento *postpass*, ou até mesmo desabilitando, ou habilitando ambos os escalonamentos para o mesmo processo de compilação.

O método padrão de escalonamento, tanto para a abordagem *prepass*, quanto para a abordagem *postpass*, é chamado de *escalonamento por região*. Todavia, é importante ressaltar que as regiões adotadas pelo LLVM não são definidas conforme a literatura [Aho et al., 2006]. Uma região no LLVM sempre pertence a um único bloco básico, e um mesmo bloco básico pode conter mais de uma região. Isso pode ser visto no método *scheduleRegions* da classe *MachineSchedulerBase*, implementado no arquivo *MachineScheduler.cpp*.

O escalonamento padrão do LLVM é, portanto, local e intraprocedimental. A Figura 5.5 apresenta o diagrama das principais classes associadas à tarefa de escalona-

mento de instruções.

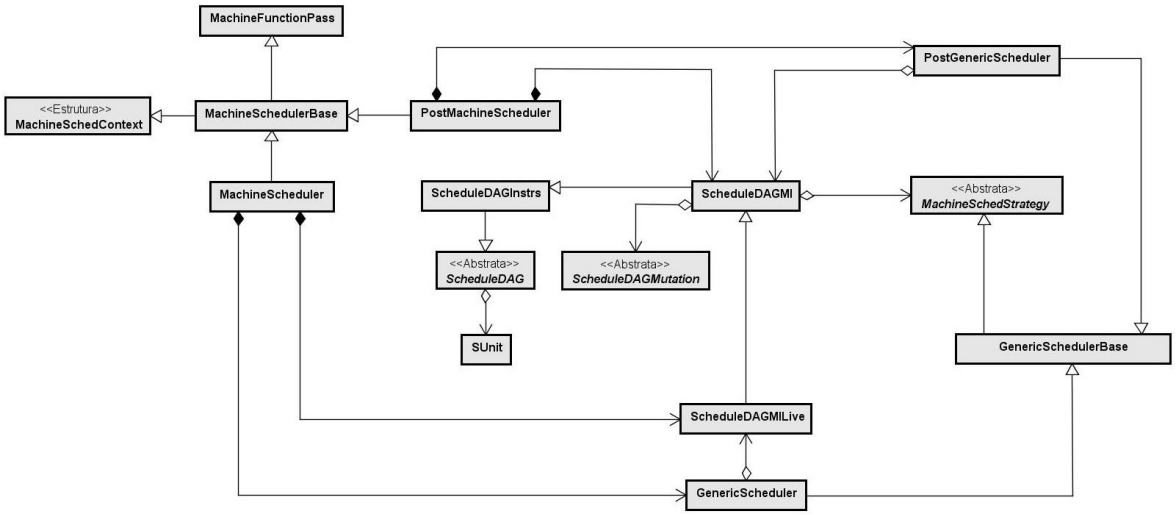


Figura 5.5. Diagrama de classes da tarefa de escalonamento de instruções.

Alocação de Registradores

No modelo padrão de geração de código, o alocador de registradores recebe as instruções do código como uma lista de instâncias da classe MachineInstr. Essas instruções podem estar escalonadas, ou não, dependendo dos parâmetros de compilação utilizados. De qualquer maneira, todas elas estarão na forma SSA. Finalizada a alocação dos registradores, nenhuma das instruções estará na forma SSA, uma vez que todas as referências aos registradores virtuais terão sido substituídas por referências aos registradores da máquina alvo, ou à memória, caso o derramamento de valores tenha ocorrido. Neste último caso, as instruções necessárias ao armazenamento e ao carregamento dos valores derramados terão sido inseridas no código objeto.

Quase todos os alocadores definidos no LLVM realizam sua tarefa considerando todos os blocos básicos da função que está sendo processada. Sendo assim, a alocação de registradores no LLVM é quase sempre global e intraprocedimental. Apenas o método *Fast* realiza alocação local e intraprocedimental. Os alocadores disponíveis no LLVM são os seguintes:

- Basic – Não é considerado um alocador para ser utilizado em ambiente de produção. Trata-se de um modelo que pode ser utilizado como estrutura básica para a construção de outros alocadores. Ele é particularmente útil para rastrear falhas e também pode ser adotado como base em comparações de desempenho. Ele define uma abordagem incremental para o processo de alocação, segundo a qual

cada pseudoregistrador é atribuído a um registrador de acordo com uma ordem determinada por uma heurística.

- Greedy – Este é o alocador padrão utilizado pelo LLVM. Trata-se de uma implementação de alto desempenho, feita a partir do alocador Basic. Ele se baseia na análise de variáveis vivas e tenta minimizar o custo do derramamento de variáveis para a memória.
- Fast – Este é o alocador padrão para as compilações feitas no modo de depuração. Esse modo é definido passando-se para o compilador o parâmetro *-O0*. Trata-se de um alocador que atua sobre cada um dos blocos básicos da função e que tenta reutilizar os registradores sempre que possível.
- PBQP – Modela o problema de alocação de registradores como um problema de otimização conhecido como PBQP (*Partitioned Boolean Quadratic Problem*), que é um caso especial do problema de atribuição quadrática (*Quadratic Assignment Problem - QAP*). Assim, a alocação dos registradores é feita com base na solução encontrada para o problema PBQP.

5.2 SPEC CPU2006

SPEC (*Standard Performance Evaluation Corporation*) é uma organização sem fins lucrativos, atualmente integrada por diferentes empresas, universidades e instituições de pesquisa, que possui o objetivo de estabelecer e manter *benchmarks* para computadores e sistemas de computação [Corporation, 2017b].

De modo geral, um *benchmark* é um padrão de medida, que pode ser adotado como referência em um processo de avaliação. Cada um dos *benchmarks* SPEC CPU é um programa executável, que realiza um conjunto específico de operações e cuja execução fornece uma medida de desempenho. Todos os conjuntos de *benchmarks* SPEC CPU têm o objetivo de avaliar o desempenho de computadores em tarefas de processamento intensivo. Eles podem ser úteis para analisar o desempenho de processadores, arquiteturas de memória e compiladores [Henning, 2006].

Execuções de um mesmo *benchmark* em computadores diferentes permitem que esses computadores sejam avaliados comparativamente. Execuções em um mesmo computador, configurado de diferentes maneiras a cada execução, permitem uma avaliação comparativa das configurações de *hardware* utilizadas. Por outro lado, múltiplas execuções, em um mesmo computador, de um *benchmark* compilado de diferentes maneiras a cada execução, permitem uma avaliação comparativa das compilações feitas.

Além dos *benchmarks*, os conjuntos SPEC CPU possuem ferramentas para a geração e apresentação de relatórios sobre as avaliações feitas, e também para a automação das tarefas de compilação e execução dos *benchmarks*. Esses relatórios, quando gerados conforme um protocolo pré-estabelecido, podem ser submetidos a SPEC, a fim de que os resultados encontrados sejam publicados.

Para possibilitar a comparação de computadores e arquiteturas diferentes, SPEC fornece os códigos fonte dos *benchmarks* e não os programas executáveis correspondentes. Esses códigos podem ser então compilados em cada uma das arquiteturas que será testada, a fim de que o executável apropriado seja gerado. Essa compilação pode ser feita externamente, mas também pode ser realizada a partir das ferramentas disponibilizadas no próprio conjunto SPEC CPU.

Uma vez que um programa pode ser compilado de diferentes maneiras para uma mesma arquitetura, duas possibilidades de compilação podem ser utilizadas: *base* e *peak*. A primeira delas impõe que, para todos os *benchmarks*, os mesmos parâmetros de compilação sejam utilizados na mesma ordem. A segunda é mais flexível e permite a utilização de parâmetros de otimização específicos para cada um dos *benchmarks*, a fim de favorecer o desempenho dos executáveis na arquitetura considerada.

Os parâmetros necessários a esses dois tipos de compilação, assim como várias opções destinadas à ferramenta de execução, devem ser informados em arquivos específicos de configuração, para os quais alguns modelos estão disponíveis na estrutura de diretórios do conjunto.

De modo particular, o conjunto SPEC CPU2006 pode ser utilizado para avaliar o *desempenho*, isto é, a *velocidade de processamento*, medindo o quão rápido uma tarefa pode ser realizada, e também a *capacidade de processamento*, mensurando o volume de tarefas que podem ser executadas simultaneamente.

Ele é formado por dois grupos: CINT2006, que agrupa *benchmarks* voltados para o processamento de números inteiros; e CFP2006, que possui *benchmarks* destinados ao processamento de números de ponto flutuante. Os *benchmarks* de cada um desses grupos estão relacionados a diferentes áreas e aplicações e foram escritos nas linguagem C, C++ e Fortran. As tabelas 5.1 e 5.2 apresentam todos os *benchmarks* dos grupos CINT2006 e CFP2006 respectivamente.

Além dos *benchmarks* listados nessas tabelas, dois *pseudo-benchmarks* existem no conjunto: *998.specrand* e *999.specrand*, ambos escritos em C. O primeiro deles pertence ao grupo CFP2006 e o segundo ao grupo CINT2006. Esses dois programas são geradores de números aleatórios e são utilizados por vários *benchmarks*. Eles foram fornecidos separadamente para auxiliar no diagnóstico de possíveis problemas durante a execução dos *benchmarks*. Eles podem ser utilizados da mesma forma que qualquer *benchmark*,

Benchmark	**Linguagem**	**Área**
400.perlbench	C	Linguagem de Programação
401.bzip2	C	Compressão de Dados
403.gcc	C	Compiladores
429.mcf	C	Otimização Combinatorial
445.gobmk	C	Inteligência Artificial
456.hmmer	C	Genética
458.sjeng	C	Inteligência Artificial
462.libquantum	C	Computação Quântica
464.h264ref	C	Compressão de Vídeo
471.omnetpp	C++	Simulação de Eventos Discretos
473.astar	C++	Algoritmo de Busca (Melhor Caminho)
483.xalancbmk	C++	Processamento XML

Tabela 5.1. *Benchmarks* SPEC CPU2006 CINT2006.

Benchmark	**Linguagem**	**Área**
410.bwaves	Fortran	Dinâmica dos Fluidos
416.gamess	Fortran	Química Quântica
433.milc	C	Física (Física de Partículas)
434.zeusmp	Fortran	Física (Dinâmica dos Fluidos)
435.gromacs	C, Fortran	Bioquímica, Dinâmica Molecular
436.cactusADM	C, Fortran	Física (Relatividade Geral)
437.leslie3d	Fortran	Dinâmica dos Fluidos
444.namd	C++	Biologia, Dinâmica Molecular
447.dealII	C++	Análise de Elementos Finitos
450.soplex	C++	Otimização, Programação Linear
453.povray	C++	Image Ray-tracing
454.calculix	C, Fortran	Mecânica Estrutural
459.GemsFDTD	Fortran	Física (Eletromagnetismo)
465.tonto	Fortran	Química Quântica
470.lbm	C	Dinâmica dos Fluidos
481.wrf	C, Fortran	Previsão Climática
482.sphinx3	C	Reconhecimento de Fala

Tabela 5.2. *Benchmarks* SPEC CPU2006 CFP2006.

mas as medições associadas a eles não são incluídas nos relatórios de resultados.

Para os *benchmarks* do grupo CINT2006, compilados com a opção *base*, os resultados das avaliações de desempenho são a média geométrica de doze medições normalizadas. Para os que pertencem ao grupo CFP2006, compilados com a mesma opção, os resultados são a média geométrica de dezessete medições normalizadas.

Os *benchmarks* podem ser executados com três conjuntos de dados de entrada: *test*, *train* e *ref*. O primeiro desses conjuntos, *test*, permite uma execução mais rápida, pois é o que possui menor volume de dados. Trata-se de um conjunto simples, que visa apenas garantir que o programa gerado é capaz de funcionar sem erros de execução. O segundo conjunto, *train*, possui tamanho intermediário e é utilizado para gerar informações que podem ser aproveitadas pelos compiladores que usam técnicas

de otimização orientadas a perfil. Essas técnicas, conhecidas como *Profile-Guided Optimizations*, ou *Feedback-Directed Optimizations*, buscam otimizar o código baseando-se em informações sobre o perfil de execução do programa, tais como tempo total de execução, quantidade de memória consumida e duração das chamadas de funções. O terceiro e último conjunto de dados de entrada, *ref*, é uma massa real de dados para a aplicação que o *benchmark* representa e é o que possui maior tamanho.

Para que um relatório de resultados possa ser submetido a SPEC, no mínimo uma *execução completa* de todos os *benchmarks* de um dos grupos do conjunto deve ser realizada. A compilação desses *benchmarks* precisa ter sido feita com a opção *base*. A execução dos programas compilados com a opção *peak* é opcional. Uma execução completa compreende os seguintes passos: ❶ uma única execução dos *benchmarks* com o conjunto de dados *test*; ❷ uma única execução com o conjunto de dados *train*; ❸ três execuções sequenciais com o conjunto de dados *ref*. Há uma ferramenta para automatizar a execução desses passos. Os resultados de cada um deles podem ser vistos nos arquivos de registro gerados (arquivos *.log*). No relatório, as medições relacionadas aos conjuntos de dados *test* e *train* não são computadas. Apenas as três medições feitas durante as execuções com o conjunto de dados *ref* são apresentadas.

5.3 Considerações Finais

Neste capítulo, informações básicas sobre o compilador LLVM e sobre o conjunto de *benchmarks* SPEC CPU2006 foram apresentadas. O LLVM foi utilizado para a implementação da solução proposta no Capítulo 4 para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Por sua vez, o SPEC CPU2006 foi utilizado para a realização dos testes da implementação feita.

O capítulo que se segue apresenta o código desenvolvido para implementar a abordagem de Pinter no LLVM, detalhando suas principais estruturas e características.

Capítulo 6

Implementação da Solução Proposta por Pinter

A abordagem de Pinter foi escolhida como solução para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Este capítulo descreve a implementação dessa abordagem no LLVM.

A Seção 6.1 descreve as decisões tomadas em relação ao código implementado e mostra as principais classes, estruturas de dados e algoritmos utilizados nessa implementação. A Seção 6.2 apresenta as considerações finais.

6.1 Implementação

A abordagem de Pinter, descrita na Seção 4.1, foi implementada na Versão 3.7.0 do LLVM. O desenvolvimento foi feito sobre a plataforma Windows® de 64 bits, em uma máquina com sistema operacional Microsoft Windows 10. Os *softwares* utilizados foram os seguintes:

- Microsoft Windows 10 Home Single Language 64 bits;
- TortoiseSVN Versão 1.9.5.27581 (64-bit);
- LLVM Versão 3.7.0 (somente os projetos *LLVM Core* e *CLang*);
- Microsoft Visual Studio Community 2015 Versão 14.0.25431.01 *update* 3;
- Microsoft .NET Framework Versão 4.7.02556;
- CMake Versão 3.7.1;
- Python Versão 3.6.0;

- ActivePerl Versão 5.22.2 *build* 2203 (64-bit);
- GnuWin32 Versão 0.6.3;
- Graphviz Versão 2.38.

Todos esses softwares são essenciais à criação, ou ao funcionamento do ambiente de desenvolvimento, exceto o último deles, Graphviz, que é necessário apenas para a visualização de certos grafos gerados pelo próprio LLVM, tais como os *SelectionDAGs* e os grafos de fluxo de controle. A visualização desses grafos foi útil durante a fase de aprendizado do LLVM, mas em nada se relaciona com o desenvolvimento feito.

Todos os códigos elaborados foram escritos em C++ e o ambiente utilizado para o desenvolvimento de todos eles foi o Microsoft Visual Studio. De acordo com a lista de softwares apresentada, apenas os projetos *LLVM Core* e *CLang* do portfólio LLVM foram necessários à implementação da solução. De fato, todas as alterações e inclusões de código foram feitas na biblioteca estática *LLVMCodeGen*, pertencente ao projeto *LLVM Core*. As alterações nessa biblioteca visaram alterar o gerador de código LLC, conforme ilustra a Figura 6.1.

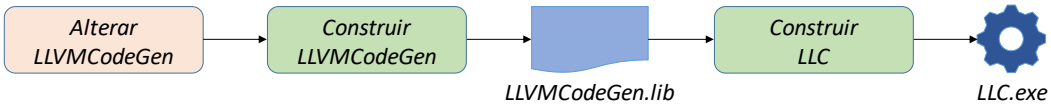


Figura 6.1. Geração do compilador LLC.

O projeto *CLang* em nada foi alterado. Conforme a explicação feita na Seção 5.1, esse projeto foi utilizado somente para a geração do *frontend* necessário à tradução dos códigos fontes para a representação intermediária do LLVM, durante a fase de testes. Na implementação realizada, a alocação de registradores é global e intraprocedimental, enquanto o escalonamento de instruções é local.

Para cada uma das funções existentes no programa fonte que está sendo compilado, as seguintes etapas sequenciais são executadas:

1. um único grafo de interferência paralelizável é construído;
2. a alocação dos registradores é realizada a partir das informações desse grafo;
3. o escalonamento das instruções é efetuado de acordo com o método padrão definido no LLVM, que ordena as instruções de cada um dos blocos básicos existentes na função.

As duas primeiras etapas são realizadas em um único passo do gerador de código, o passo que realiza a alocação dos registradores. A terceira e última etapa é executada no passo que escalona as instruções.

Todos os códigos desenvolvidos estão disponíveis em `http://bit.ly/2F3e9q5`, juntamente com todas as ferramentas construídas para automatizar tarefas das fases de implementação e teste. Dentre essas ferramentas, encontram-se aquelas destinadas à criação do ambiente de desenvolvimento, elaboradas conforme os passos gerais de instalação do LLVM, descritos em Team [2017].

6.1.1 Desenvolvimento do Código

As seguintes classes foram construídas para a implementação da abordagem de Pinter no LLVM:

- FunctionData
- Graph
- InterferenceGraph
- SchedulingGraph
- ParallelizableGraph
- PinterStrategy
- IGraphPinter
- RegAllocPinter

As cinco primeiras estão associadas à construção do grafo de interferência paralelizável. As três últimas estão relacionadas à implementação do alocador de registradores que utiliza esse grafo.

Além dessas classes, quatro outras também foram utilizadas. Elas foram desenvolvidas para o trabalho de Silva [2015] e foram reutilizadas neste trabalho após as modificações necessárias. Elas estão descritas na Seção 6.1.1.8 e são as seguintes: *ColoringStrategy* (classe abstrata), *GeorgeStrategy*, *RegisterClassTree* e *IGraph*.

A Figura 6.2 apresenta o diagrama de classes do código desenvolvido para implementar a abordagem de Pinter. Nesse diagrama, as estruturas e classes do LLVM que foram herdadas pelas classes construídas também são apresentadas. As seções que se seguem a essa figura trazem as principais informações sobre as classes construídas e reutilizadas, a fim de facilitar o entendimento do código desenvolvido.

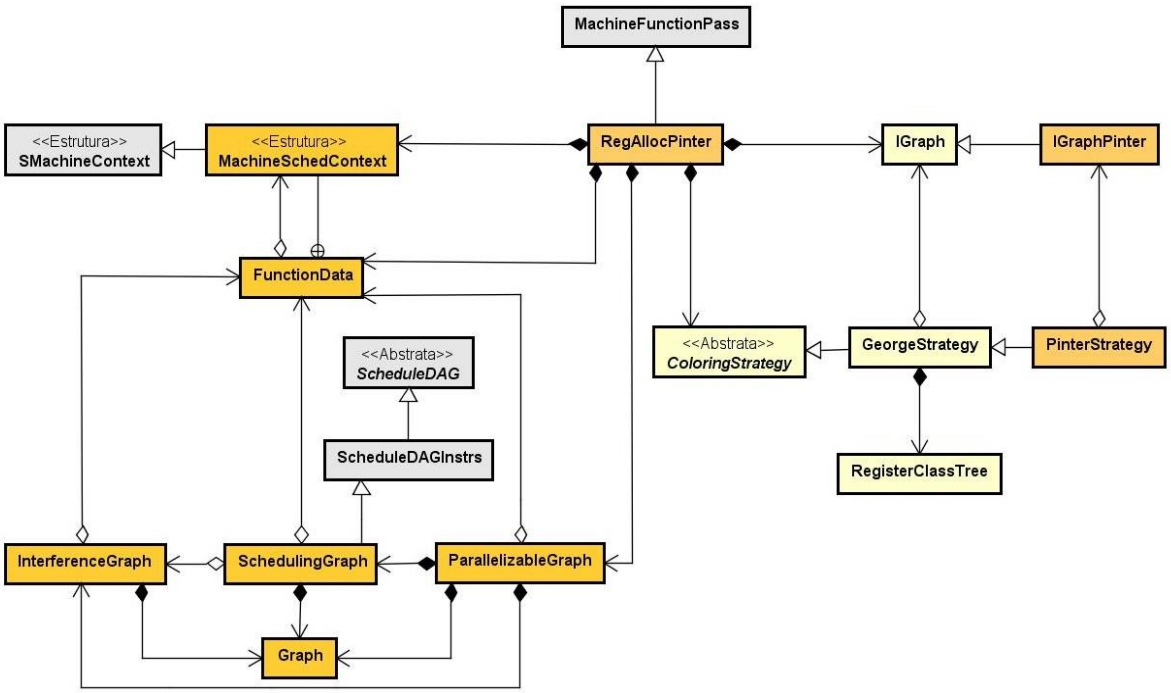


Figura 6.2. Diagrama das principais classes utilizadas para implementar a abordagem de Pinter.

6.1.1.1 Classe FunctionData

A principal finalidade da classe FunctionData é a construção das estruturas que armazenam informações sobre as instruções da função que está sendo processada. Ao todo, três estruturas existem para essa finalidade: ❶ SFuncData, que contém as instâncias MachineInstr, os índices e os números dos blocos básicos de cada instrução; ❷ SBBlockData, que armazena os dados dos blocos básicos da função, permitindo a identificação dos sucessores de cada um dos blocos; e ❸ SRegion que agrupa os dados necessários à utilização dos algoritmos do LLVM capazes de criar um grafo de escalonamento a partir de um determinado conjunto de instruções. A Figura 6.3 mostra os relacionamentos entre essas entidades, desconsiderando o fato de todas as estruturas serem aninhadas em relação à classe.

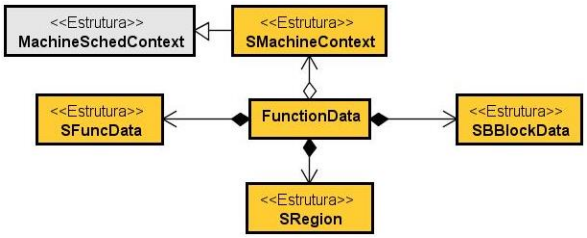


Figura 6.3. Estruturas utilizadas pela classe FunctionData.

A classe FunctionData também é responsável pelo armazenamento de objetos de contexto, determinados pelo próprio código que está sendo compilado e também pela arquitetura para a qual a compilação está sendo feita. Esses objetos são armazenados na estrutura SMachineContext.

6.1.1.2 Classe Graph

Esta é uma classe básica, construída para a manipulação de grafos simples, multigrafos e pseudografos, sejam eles dirigidos, ou não dirigidos. Um grafo é dito simples se não possui arestas paralelas nem laços. Arestas paralelas são aquelas que unem o mesmo par de vértices e laços são arestas que conectam um vértice a ele mesmo. Um grafo que possui ao menos um laço é um pseudografo. Um grafo que contém arestas paralelas é um multigrafo.

Trata-se de uma classe totalmente independente das estruturas e algoritmos existentes no LLVM, exceto pela utilização das macros DEBUG(), que permitem a impressão de mensagens de depuração, quando o gerador de código é executado com parâmetros apropriados. Entretanto, essas macros podem ser facilmente identificadas, desabilitadas, ou mesmo eliminadas do código, permitindo que a classe seja reutilizada em qualquer outro projeto que necessite da manipulação de grafos.

A classe implementa várias funcionalidades para a manipulação de grafos e conta com vários métodos de inserção, exclusão e localização de vértices e arestas. Para representar um grafo, ela utiliza uma matriz de adjacência e também pode utilizar listas de adjacências simultaneamente. Possui métodos de busca em profundidade e busca em largura, e possibilita que as funções que manipulam os vértices do grafo em cada uma dessas buscas sejam indicadas por meio de ponteiros. Ela permite a habilitação e a desabilitação da alocação física dos vértices e arestas do grafo, a fim de atender requisitos de desempenho e memória. De forma mais relacionada à essência da abordagem, a classe contém métodos para a conversão de DAGs em grafos não dirigidos, e métodos que definem o complemento e o fechamento transitivo de um grafo. Dois algoritmos para o fechamento transitivo de um grafo foram implementados: o algoritmo de *Floyd-Warshall*, que utiliza programação dinâmica, e um algoritmo baseado em busca em largura. Este último mostrou-se bem mais eficiente, uma vez que não determina os caminhos entre todos os pares de vértices do grafo. Ele utiliza as listas de adjacências para verificar caminhos somente entre os pares de vértices que podem ser conectados. A escolha do algoritmo a ser utilizado é feita por meio de um parâmetro existente no método.

6.1.1.3 Classe InterferenceGraph

Essa classe cria o grafo de interferência associado à função que está sendo processada, a partir da análise global dos intervalos de vida dos registradores virtuais utilizados no código.

Dois tipos de grafos de interferência podem ser gerados. No primeiro, os vértices do grafo representam os registradores virtuais utilizados no código. Trata-se do grafo de interferência tradicional. No segundo, os vértices representam as instruções da função nas quais algum pseudoregistrador é definido. Logicamente, nesse caso, os vértices do grafo também estão associados aos registradores virtuais utilizados, mas de maneira indireta. Para ilustrar a diferença entre esses dois tipos de grafos, considere o *bitcode* da Figura 6.4.

```
%vreg1<def> = MOV32rm <fi#-2>, 1, %noreg, 0, %noreg
%vreg0<def> = MOV32rm <fi#-1>, 1, %noreg, 0, %noreg
%vreg3<def> = MOV32rm %vreg0, 1, %noreg, 0, %noreg
%vreg3<def,tied1> = SUB32rm %vreg3<tied0>, %vreg1, 1
MOV32mr %vreg0, 1, %noreg, 0, %noreg, %vreg3
CMP32ri8 %vreg3, 2, %EFLAGS<imp-def>
MOV32mr %ESP, 1, %noreg, 4, %noreg, %vreg1
MOV32mr %ESP, 1, %noreg, 0, %noreg, %vreg0
CMP32mi8 %vreg0, 1, %noreg, 0, %noreg, 1, %EFLAGS<imp-def>
RETL
```

Figura 6.4. Fragmento de um *bitcode* gerado pelo LLVM.

Esse código utiliza três registradores virtuais, *%vreg0*, *%vreg1* e *%vreg3*, e as suas primeiras quatro instruções definem esses pseudoregistradores. A análise de variáveis vivas indica que o grafo de interferência tradicional será um grafo completo de três vértices, como indica a Figura 6.5 (a). Por sua vez, o grafo de interferência cujos vértices representam as instruções, possuirá quatro vértices, um para cada uma das instruções que definem os registradores virtuais, como mostra a Figura 6.5 (b). É fácil verificar que, a partir desse último grafo, o grafo de interferência tradicional pode ser derivado. O contrário, contudo, não pode ser feito.

O grafo de interferência cujos vértices representam as instruções do código é essencial para a implementação da solução e foi sugerido em Pinter [1996]. De fato, ele viabiliza a união do grafo de falsas dependências, gerado a partir do grafo de escalonamento, cujos vértices também são instruções do programa, com o grafo de interferência, para a criação do grafo de interferência paralelizável. Além disso, ele simplifica a construção do grafo de escalonamento, reduzindo o número de instruções que precisam ser consideradas.

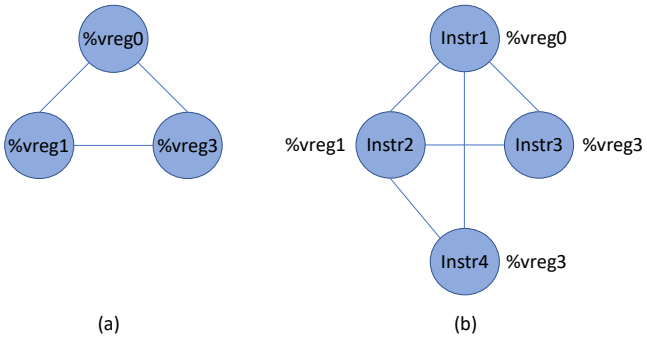


Figura 6.5. Grafos de interferência gerados pela classe InterferenceGraph: (a) grafo tradicional e (b) grafo cujos vértices representam instruções do programa.

6.1.1.4 Classe SchedulingGraph

A classe SchedulingGraph cria o DAG de escalonamento objetivando a geração do grafo de falsas dependências. Ela também identifica as restrições de máquina que devem ser impostas ao código, em função das características da arquitetura alvo. Essas restrições são armazenadas como arestas, que serão utilizadas posteriormente para a geração do grafo de falsas dependências.

O processo de criação do DAG de escalonamento utiliza as mesmas estruturas e algoritmos básicos definidos no LLVM para a realização do escalonamento de instruções. Esse processo pode ser dividido em duas etapas: ❶ criação e escalonamento do *SelectionDAG* e ❷ construção do DAG de escalonamento a partir do *SelectionDAG* escalonado.

Para a realização da primeira etapa, os algoritmos de construção e escalonamento do *SelectionDAG* foram estendidos, de modo a considerarem todo o código da função em análise. A principal estrutura de dados do *SelectionDAG* é um vetor de objetos da classe *SUnit*. Cada um desses objetos armazena uma única instrução do código, ou seja, uma instância da classe *MachineInstr*, e também dados associados ao sequenciamento dessa instrução. Em função de um grande esforço para que o código básico do LLVM fosse minimamente alterado, a construção dessa etapa exigiu somente as seguintes modificações no código original[1]:

- No arquivo *ScheduleDAGInstrs.h*: os métodos *initSUnits*, *addSchedBarrierDeps* e *buildSchedGraph* foram declarados virtuais;

[1]Com exceção das modificações descritas na Seção 6.1.2, sem as quais a abordagem de Pinter não poderia ser incluída como um novo passo do gerador de código, essas foram as únicas modificações feitas no código original do LLVM.

- No arquivo *ScheduleDAGInstrs.cpp*: as funções *getUnderlyingObjectsForInstr*, *adjustChainDeps*, *addChainDependency* e *isGlobalMemoryObject* deixaram de ser estáticas.

Na segunda etapa, o DAG de escalonamento é construído a partir da avaliação de cada um dos objetos *SUnit* do *SelectionDAG*. Conforme a definição apresentada na Seção 4.1 para o grafo de interferência paralelizável, o DAG de escalonamento precisa possuir somente as instruções que foram incluídas no grafo de interferência criado pela classe *InterferenceGraph*. Por essa razão, os vértices desses dois grafos são sempre os mesmos. Ainda conforme a Seção 4.1, as arestas de controle de fluxo do programa não são incluídas no DAG. A Figura 6.6 apresenta o grafo de escalonamento gerado para o *bitcode* apresentado na Figura 6.4.

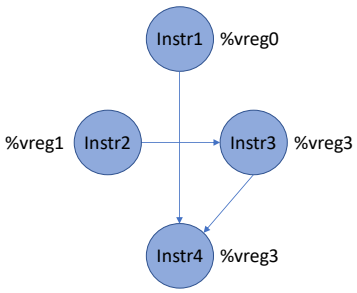


Figura 6.6. Grafo de escalonamento gerado pela classe SchedulingGraph.

Idealmente, a determinação dos blocos básicos que podem ser escalonados simultaneamente deveria ser feita nessa classe, utilizando análises de dominância e pós-dominância, conforme sugerido em Pinter [1996]. Assim como foi feito com as restrições de máquina, arestas entre as instruções dos blocos que não devem ser escalonados em paralelo poderiam ser armazenadas, para serem utilizadas após a retirada das direções das arestas do grafo gerado pelo fechamento transitivo do grafo de escalonamento. Todavia, as análises de dominância e pós-dominância não foram implementadas. Como resultado, o grafo de interferência paralelizável pode demandar um número maior de registradores e dificultar o processo de alocação. Teoricamente, a tarefa de escalonamento de instruções não é prejudicada nem beneficiada em função disso.

A Figura 6.7 ilustra essa situação. Nessa figura, (a) é um CFG, um grafo de fluxo de controle, enquanto (b) e (c) ilustram grafos de interferência paralelizável construídos a partir desse CFG. A geração do grafo em (b) considerou análises de dominância e pós dominância. A geração do grafo em (c) não considerou essas análises.

É necessário ressaltar que a ausência dessas análises não invalida a implementação feita, afetando de forma caótica os resultados dela decorrentes, fazendo com que tais

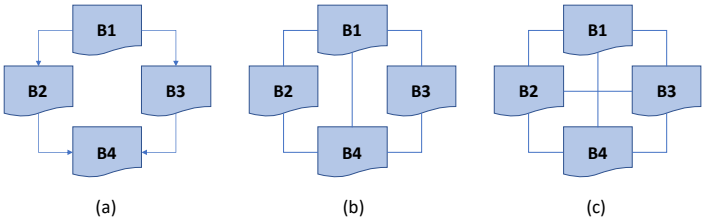


Figura 6.7. (a) CFG; (b) e (c) grafos de interferência paralelizável para o CFG em (a), gerados, respectivamente, considerando e desconsiderando análises de dominância e pós dominância.

resultados sejam bons em certos casos e ruins em alguns outros de forma imprevisível. Realmente, como o grafo de interferência paralelizável gerado sem a consideração dessas análises é sempre mais restritivo, os resultados da implementação feita podem ser vistos como um limite inferior para os resultados de quaisquer outras implementações que considerem essas análises e a mesma arquitetura alvo.

6.1.1.5 Classe ParallelizableGraph

Essa classe constrói o grafo de falsas dependências e também o grafo interferência paralelizável. Assim como os grafos de interferência e o grafo de escalonamento descritos anteriormente, esses dois grafos são objetos do tipo *Graph*. Todos os principais métodos de mais alto nível envolvidos no processo de construção desses grafos podem ser vistos no código apresentado no fim desta seção.

Conforme indicado no método *build* desse código, o objeto *Graph* da classe pode ser o grafo de interferência paralelizável, ou então, caso o atributo PGraphIsIGraph seja verdadeiro, o grafo de interferência tradicional. Essas duas possibilidades foram consideradas nos testes realizados.

A construção do grafo de interferência paralelizável segue os passos indicados na Seção 4.1. O último desses passos define que o conjunto de arestas do grafo de interferência paralelizável é formado pela união das arestas do grafo de interferência com as arestas do grafo de falsas dependências. Assim, na implementação feita, em conformidade com a sugestão feita em Pinter [1996], todas as arestas de interferência[2] sempre são inseridas no grafo de interferência paralelizável, e as arestas de falsas dependências[3] são inseridas de acordo com uma das quatro estratégias que se seguem:

1. **ALL** - Todas as arestas de falsas dependências são inseridas. Nenhuma aresta é descartada.

[2]Arestas que pertencem ao grafo de interferência.
[3]Arestas que pertencem ao grafo de falsas dependências.

2. **DEFAULT** - Uma aresta de falsa dependência é inserida, se, para cada um dos vértices sobre os quais ela incide, o grau do vértice permanecer menor do que o número de registradores disponíveis para ele, considerando a possível inserção da aresta.
3. **LIMIT** - Uma aresta de falsa dependência é inserida, se, para cada um dos vértices sobre os quais ela incide, o grau do vértice permanecer menor do que o número total de registradores disponíveis para todos os vértices do grafo, considerando a possível inserção da aresta.
4. **NONE** - Nenhuma aresta de falsa dependência é inserida. Nesta estratégia, o grafo de interferência paralelizável é exatamente igual ao grafo de interferência.

A escolha da estratégia que deve ser adotada é feita definindo-se um dos atributos da classe. As estratégias foram definidas principalmente porque, nos casos de *bitcodes* médios ou grandes, o número total de arestas de falsas dependências pode atingir a ordem dos milhões e, nesses casos, se todas elas forem inseridas no grafo de interferência paralelizável, a tarefa de alocação de registradores nem sempre é capaz de chegar a uma solução em tempo viável.

Em Pinter [1996], uma estratégia para a alocação de registradores foi sugerida. Nela, durante a fase de alocação, arestas de falsas dependências são removidas dos vértices do grafo de interferência paralelizável cujos graus, considerando-se apenas as arestas de interferência, são menores do que o número total de registradores disponíveis. Na estratégia DEFAULT implementada, ao invés de serem inseridas, para posteriormente, no momento da alocação, serem removidas, as arestas de falsas dependências são inseridas no grafo observando a mesma condição definida por Pinter. Esse procedimento produz o mesmo efeito da estratégia sugerida por Pinter e traz o benefício adicional de diminuir o tempo necessário para a construção do grafo de interferência paralelizável.

Com relação à criação do grafo de falsas dependências, é interessante dizer que a operação de *complemento*, indicada no Passo 5 da Seção 4.1, não foi feita com o método correspondente implementado na classe *Graph*. A complexidade do algoritmo implementado é $O(n^2)$, sendo n o número de vértices do grafo, de tal modo que se trata de um algoritmo comum para realizar essa operação. Por essa razão, ele chegou a ser testado durante a fase de desenvolvimento, mas, por fim, a sua utilização foi descartada, a fim de melhorar o desempenho do processo de construção do grafo de falsas dependências. Em substituição a essa operação, as arestas de falsas dependências inseridas no grafo de interferência paralelizável são aquelas que complementam o grafo de falsas dependências que foi construído, mas que não existem nele de fato.

... Construção do Grafo de Interferência Paralelizável.cpp 1

```
1  void ParallelizableGraph::build(bool CreateSortedVectors) {
2      if (PGraphIsIGraph) {
3          buildInterferenceGraph(InterferenceGraph::VREGS);
4          assert(I != nullptr &&  "*** build: interference graph must exists.");
5          P.reset(new Graph(I->getGraph()));
6          P->setName("Parallelizable Graph (Interference Graph) for function " +
7              FunctionName);
8          TotalInterferenceEdges = I->getInterferenceEdges().size();
9          TotalFalseDependenceEdges = 0;
10         if (!KeepGraphs)
11             I.reset(nullptr);
12     }
13     else {
14         buildInterferenceGraph();
15         buildSchedulingGraph();
16         buildFalseDependenceGraph();
17         buildParallelizableGraph();
18     }
19     if (CreateSortedVectors)
20         createSortedVectors();
21 }
22
23 void ParallelizableGraph::buildInterferenceGraph(
24     TInterferenceGraphVertexTypes Type) {
25     I.reset(new InterferenceGraph(FD, Type));
26     if (!KeepGraphs) {
27         if (Type == InterferenceGraph::POSITIONS)
28             I->disableAllAllocations();
29         else if (Type == InterferenceGraph::VREGS)
30             I->setAllocateEdges(false);
31     }
32     I->build();
33 }
34
35 void ParallelizableGraph::buildSchedulingGraph() {
36     assert(I != nullptr &&
37         "*** buildSchedulingGraph: interference graph must exists.");
38     S.reset(new SchedulingGraph(FD, *I));   // All allocations are necessary
39     S->build();
40 }
41
42 void ParallelizableGraph::buildFalseDependenceGraph() {
43     assert(S != nullptr &&
44         "*** buildFalseDependenceGraph: scheduling graph must exists.");
45     F.reset(new Graph(S->getGraph()));
46     F->changeToTransitiveClosure();
47     F->changeToSubjacent(true);
48     TotalPossibleFDEdges = F->maxEdges() - F->getTotalEdges();
49     if (!KeepGraphs) {
50         F->disableEdgesAllocation(true);
51         F->disableAdjacencyListAllocation(true);
52     }
53     addMachineConstraints();
54     addControlDependences();
55     TotalPossibleFDEdges -= S->getMachineConstraints().size();
56     TotalPossibleFDEdges -= S->getControlDependences().size();
```

... Construção do Grafo de Interferência Paralelizável.cpp 2

```
57      F->setName("False Dependence Graph Without Complement for function " +
58          FunctionName);
59      if (!KeepGraphs) {
60          S.reset(nullptr);   // Scheduling graph is no longer necessary
61      }
62  }
63
64  void ParallelizableGraph::buildParallelizableGraph() {
65      assert(I != nullptr &&
66          "*** buildParallelizableGraph: interference graph must exists.");
67      assert(F != nullptr &&
68          "*** buildParallelizableGraph: false dependence graph must exists.");
69
70      // I and F graphs must have the same number and the same order of vertices
71      // See construction of scheduling graph
72      assert(I->getPositions().size() == F->getTotalVertices() &&
73          "*** buildParallelizableGraph: different number of vertices");
74
75      // The Parallelizable Graph
76      // Vertices are VRegs
77      P.reset(new Graph());
78      P->setType(TTypes::NDAG);
79      P->setName("Parallelizable Graph for function " + FunctionName);
80      if (!KeepGraphs)
81          P->disableEdgesAllocation(true);
82
83      // Inserting the vertices into parallelizble graph
84      insertVertices();
85
86      // Inserting the interference edges into parallelizble graph
87      insertInterferenceEdges();
88
89      // Getting the physical registers that can be associated to each
90      // virtual register in the graph vertices
91      if (Strategy == DEFAULT)
92          buildPRegsLists(P->getVertices());
93
94      // Inserting the edges into parallelizable graph
95      insertFalseDependenceEdges();
96      if (KeepGraphs)
97          assert(TotalInterferenceEdges +
98              TotalFalseDependenceEdges == P->getTotalEdges() &&
99              "*** buildParallelizableGraph: invalid number of edges.");
100
101     if (!KeepGraphs) {
102         I.reset(nullptr);
103         F.reset(nullptr);
104     }
105 }
106
```

6.1.1.6 Classes PinterStrategy e IGraphPinter

Essas classes implementam duas heurísticas para a coloração do grafo de interferência paralelizável: a primeira, H1, está relacionada ao grau dos vértices do grafo, a segunda, H2, associa-se ao custo de derramamento dos pseudoregistradores representados pelos vértices do grafo. Ambas consideram aspectos do processo de alocação de registradores do LLVM e são formadas por quatro etapas, sendo as etapas 1, 2 e 4 idênticas, e a Etapa 3 aquela que as diferencia. Essas etapas são as seguintes:

- H.1 (*comum*) - Retirar do grafo os vértices que não podem ser derramados para a memória. Aqueles com custo de derramamento igual à constante *llvm::huge_ valf*.
- H.2 (*comum*) - Retirar do grafo os *vértices livres*, aqueles que não se conectam a nenhum outro vértice.
- H1.3 (*apenas para a heurística* H1) - Retirar do grafo o vértice com maior grau e menor número de registradores associados.
- H2.3 (*apenas para a heurística* H2) - Retirar do grafo o vértice com maior custo de derramamento e menor número de registradores associados.
- H.4 (*comum*) - Sempre que um vértice é retirado do grafo, o grau dos demais vértices e o número de registradores associados a cada um deles é atualizado. Os vértices cujo grau é menor do que o número de registradores são selecionados para serem coloridos. Caso o vértice retirado do grafo não possa ser atribuído a nenhum registrador, ele deve ser derramado para a memória.

A classe PinterStrategy cria os objetos necessários à execução da abordagem de Pinter, enquanto a classe IGraphPinter implementa as etapas das heurísticas apresentadas.

6.1.1.7 Classe RegAllocPinter

Essa classe implementa o alocador de registradores baseado na abordagem de Pinter. Ela cria uma instância da classe *ParallelizableGraph*, dispara o processo de construção do grafo de interferência paralelizável e realiza o processo de alocação dos registradores utilizando esse grafo.

6.1.1.8 Classes Reutilizadas

São as classes *ColoringStrategy*, *GeorgeStrategy*, *RegisterClassTree* e *IGraph*, desenvolvidas originalmente para o trabalho de Silva [2015] e reutilizadas neste trabalho.

A primeira delas é uma classe abstrata, que funciona como uma verdadeira interface para permitir o polimorfismo entre diferentes métodos de alocação de registradores [Stroustrup, 2013; Mizrahi, 2006, 1994]. As demais implementam métodos básicos para a manipulação dos tipos de dados do LLVM e também uma estratégia de coloração de grafos baseada no método descrito em [Briggs, 1992]. Essa estratégia não realiza a etapa de *coalesce*, uma vez que o LLVM já efetua o *coalescing* dos registradores virtuais antes de executar o passo de alocação de registradores. Ela utiliza uma heurística, HB, para retirar vértices do grafo considerado no processo de alocação. Essa heurística é bastante similar à heurísitica H2 descrita na Seção 6.1.1.6. Ela pode ser descrita pelas etapas que se seguem:

- HB.1 - Retira do grafo todos os vértices cujo grau é menor do que o número de registradores disponíveis.
- HB.2 - Restando vértices no grafo, o vértice com maior custo de derramamento é retirado.
- HB.3 - Sempre que um vértice é retirado do grafo, o grau dos demais vértices e o número de registradores associados a cada um deles é atualizado. Os vértices cujo grau é menor do que o número de registradores são selecionados para serem coloridos. Caso o vértice retirado do grafo não possa ser atribuído a nenhum registrador, ele deve ser derramado para a memória.

6.1.2 Criação do Passo para a Inserção do Código Desenvolvido

A abordagem de Pinter foi implementada no LLVM como um novo alocador de registradores. Esse novo alocador foi definido como um novo passo do gerador de código, seguindo o padrão de definição apresentado em Team [2017] e verificado no código dos alocadores padrão existentes no LLVM. A Figura 6.8 apresenta a estrutura utilizada para a criação e registro desse novo passo de alocação.

Basicamente, o alocador de registradores foi definido e registrado como um passo do tipo *MachineFunctionPass*, que opera sobre cada uma das funções existentes no *bitcode*. Cada uma dessas funções corresponde a uma função do código fonte. O método sobrescrito *runOnMachineFunction* é utilizado pelo gerador de código para disparar a execução do passo.

```
1   static RegisterRegAlloc pinterRegAlloc("pinter", "Pinter register allocator",
2       createPinterRegisterAllocator);
3
4   FunctionPass* llvm::createPinterRegisterAllocator() {
5       return new RegAllocPinter();
6   }
7
8   class RegAllocPinter final : public MachineFunctionPass {
9   private:
10      ...
11
12  public:
13      ...
14
15      bool runOnMachineFunction(MachineFunction &mf) override;
16      static char ID;
17  }
18
19  char RegAllocPinter::ID = 0;
20
21  bool RegAllocPinter::runOnMachineFunction(MachineFunction &mf) {
22      ...
23  }
```

Figura 6.8. Estrutura do alocador de registradores de Pinter no LLVM.

Uma série de parâmetros e estatísticas foram criados para, respectivamente, configurar e analisar o funcionamento do alocador. Dentre os parâmetros, os mais relevantes são os seguintes:

- pgraph-strategy - Define a estratégia para a inserção das arestas de falsas dependências no grafo de interferência paralelizável. Tratam-se das estratégias descritas na Seção 6.1.1.5. O valor padrão corresponde à estratégia DEFAULT.
- pgraph-tcmethod - Define o método que deve ser utilizado para a realização do fechamento transitivo do grafo de escalonamento, conforme indicado na Seção 6.1.1.2. O valor padrão corresponde ao método que se baseia na busca em largura do grafo.
- pgraph-keepgraphs - Indica se os vértices e arestas dos grafos utilizados durante o processo de alocação precisam ser mantidos na memória. Esta opção permite a redução da quantidade de memória utilizada, o que se mostrou importante durante a execução de determinados testes. O valor padrão é *falso*, ou seja, os dados do grafo podem ser eliminados da memória assim que possível.
- pinter-pgraphisigraph - Faz com que o grafo de interferência tradicional seja considerado o grafo de interferência paralelizável. Esta opção permite a execução das linhas de código 3 a 11 apresentadas no código da Seção 6.1.1.5. A sua utilização

produz o mesmo efeito prático da estratégia NONE, definida para a inserção de arestas de falsas dependências no grafo de interferência paralelizável. Todavia, enquanto a estratégia NONE gera o grafo de interferência tradicional utilizando todos os algoritmos da classe *ParallelizableGraph*, esta opção faz com que o grafo de interferência tradicional seja construído mais rapidamente, utilizando somente os métodos da classe *InterferenceGraph*. O valor padrão é *falso*, ou seja, o grafo de interferência tradicional não deve ser tomado como o grafo de interferência paralelizável.

- pinter-usegeorgestrategy - Determina que a heurística HB descrita na Seção 6.1.1.8 seja utilizada pelo alocador de registradores. Este parâmetro tem prioridade sobre o parâmetro pinter-strategy descrito a seguir. O valor padrão é *falso*.
- pinter-strategy - Define a heurística que deve ser utilizada para a alocação de registradores. Tratam-se das heurísticas H1 e H2 descritas na Seção 6.1.1.6. Independentemente do valor deste parâmetro, caso o parâmetro pinter-usegeorgestrategy, descrito no item anterior, seja *verdadeiro*, a heurística HB será utilizada. O valor padrão corresponde à heurística H2.

6.2 Considerações Finais

Neste capítulo, o código desenvolvido para implementar a abordagem de Pinter no LLVM foi descrito em termos gerais.

O código implementado é formado basicamente por doze classes escritas em C++, sendo que quatro delas foram originalmente desenvolvidas para o trabalho de Silva [2015] e reutilizadas neste trabalho após as modificações necessárias. Esse código foi desenvolvido na Versão 3.7.0 do LLVM para a plataforma Windows® de 64 bits. A ferramenta de desenvolvimento utilizada foi o Microsoft Visual Studio.

As ferramentas utilizadas para a criação do ambiente de desenvolvimento, assim como todo o código elaborado e descrito neste capítulo estão disponíveis no repositório `http://bit.ly/2F3e9q5`. Ao todo, desconsiderando as classes que foram reutilizadas, 5.775 linhas de código foram programadas.

O próximo capítulo é dedicado aos testes que foram realizados com a implementação desenvolvida. Ele apresenta e analisa os resultados obtidos a partir da execução dos *benchmarks* utilizados.

Capítulo 7

Testes e Análise dos Resultados

A implementação da abordagem de Pinter [1996] descrita no Capítulo 6 foi testada a partir de uma série de experimentos utilizando *benchmarks* amplamente difundidos em indústrias e centros de pesquisa [Corporation, 2017b]. Os *benchmarks* utilizados nos testes descritos neste capítulo pertencem ao conjunto SPEC CPU2006.

A maior parte dos resultados apresentados neste capítulo baseia-se nas medições obtidas com as ferramentas do conjunto de *benchmarks* utilizado. Contudo, alguns resultados, especialmente aqueles relacionados ao número de derramamentos para a memória, foram obtidos por meio de estatísticas implementadas no próprio código desenvolvido.

7.1 Preparação dos Testes

Os testes do código implementado foram realizados em uma máquina com a seguinte configuração básica:

- Processador: Intel Core i7-4500U 1,80 GHz com base na arquitetura x64.
- Memória RAM: 8 GB.
- Disco rígido: 1 TB com velocidade de rotação de 5.400 rpm.
- Sistema Operacional: Windows 10 Home Single Language de 64 bits.

Trata-se da mesma máquina utilizada para o desenvolvimento do código, assim, os softwares nela instalados são os mesmos que foram listados na Seção 6.1. Durante a realização dos testes, todas as interfaces de rede da máquina foram desabilitadas, e todos os serviços não essenciais do sistema operacional foram desativados, bem como o software de antivírus.

Dos dezenove *benchmarks* SPEC CPU2006 escritos exclusivamente em C ou C++, dez foram utilizados nos testes. A Tabela 7.1 apresenta os *benchmarks* utilizados.

Benchmark	**Grupo**	**Linguagem**
401.bzip2	CINT2006	C
429.mcf	CINT2006	C
433.milc	CFP2006	C
445.gobmk	CINT2006	C
458.sjeng	CINT2006	C
462.libquantum	CINT2006	C
464.h264ref	CINT2006	C
470.lbm	CFP2006	C
482.sphinx3	CFP2006	C
473.astar	CINT2006	C++

Tabela 7.1. *Benchmarks* utilizados nos testes.

Além desses *benchmarks*, os dois *pseudo-benchmarks* existentes no conjunto SPEC CPU2006, *998.specrand* e *999.specrand*, também foram utilizados nos testes. Todos os *benchmarks* associados à linguagem Fortran foram desconsiderados, uma vez que o LLVM não possui suporte para essa linguagem.

7.1.1 Compilação e Execução dos *Benchmarks*

Alguns ajustes no código de diferentes *benchmarks* precisaram ser feitos para que eles pudessem ser utilizados nos testes. Todos esses ajustes ocorreram devido às implementações específicas que existem no Windows para funções e constantes definidas pelos padrões IEEE POSIX (*Portable Operating System Interface*).

Algumas funções definidas no padrão POSIX.1, como, por exemplo, *open*, *close*, *read*, *write*, *filename*, *setmode*, *wopen* e outras, foram descontinuadas pela Microsoft. Isso porque a implementação das linguagens C e C++ no Windows segue o padrão *ISO International Standard ISO/IEC 14882*, genericamente chamado ISO C++, que define que os nomes de quaisquer entidades que não pertençam à biblioteca C padrão e que estejam implementadas no *namespace* global devem ser iniciados por um caractere sublinha [Corporation, 2017a; Josey, 2017].

Em decorrência disso, no Windows, algumas funções POSIX.1 foram substituídas por funções homônimas, mas com um caractere sublinha na frente. A função *open*, por exemplo, passou a chamar-se *_open*, *close* tornou-se *_close*, *read* virou *_read* e assim por diante. Os nomes de constantes e variáveis também sofreram essa modificação.

Além desses ajustes, alguns acertos nos parâmetros de compilação dos *benchmarks* também foram feitos. Esses parâmetros são específicos para cada um dos programas e

estão descritos na documentação de cada um deles. Basicamente, foram acrescentados os parâmetros que indicavam que a compilação seria feita para a plataforma Windows.

Nenhum dos ajustes realizados tiveram consequências sintáticas ou semânticas. Nenhum deles alterou a lógica ou o fluxo de controle dos programas. Os *benchmarks* que não puderam ser ajustados dessa maneira foram desconsiderados.

Para cada um dos *benchmarks* utilizados nos testes, oito programas executáveis foram gerados, conforme ilustra a Figura 7.1. Cada um desse programas está relacionado a uma estratégia de compilação específica.

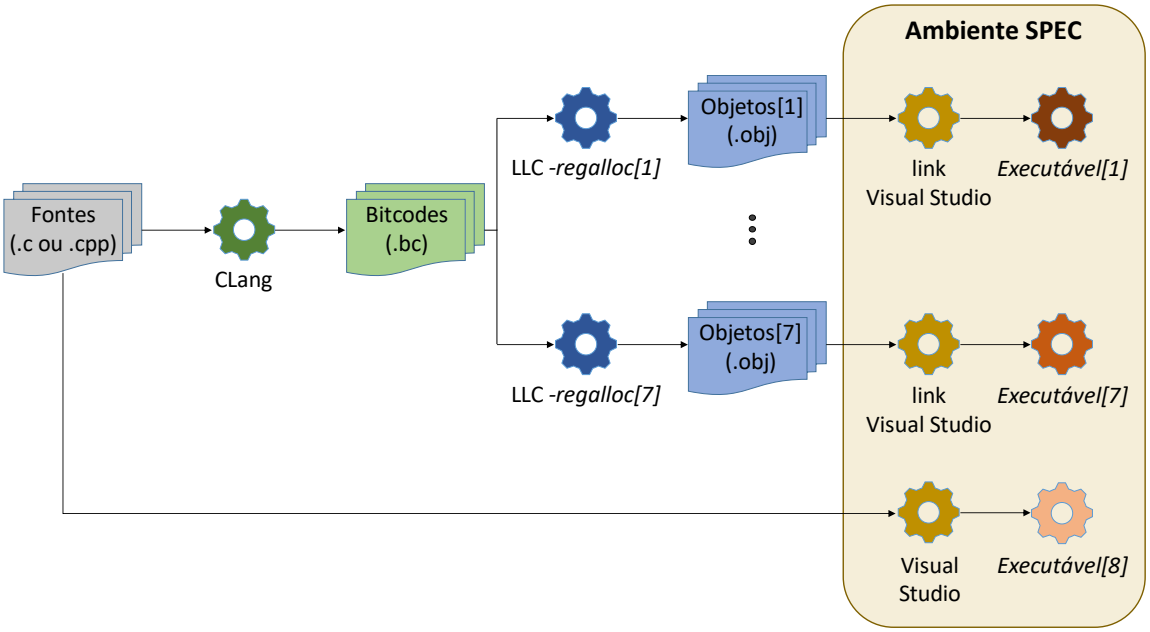


Figura 7.1. Geração dos programas executáveis para um *benchmark*.

Para sete desses oito programas, os códigos objeto foram gerados pelo LLVM e os executáveis pelo Visual Studio. Para o último desses programas, todo o processo de compilação foi realizado pelo Visual Studio.

A geração dos códigos objeto pelo LLVM foi feita fora do ambiente SPEC, para a arquitetura alvo X86. Por sua vez, a amarração dos códigos objeto gerados, utilizando o Visual Studio para a criação dos executáveis, foi realizada dentro desse ambiente. Os testes com os executáveis gerados também foram realizados dentro desse ambiente.

Todas as ferramentas desenvolvidas para a automação dos processos de compilação dos *benchmarks* e execução dos testes podem ser encontradas, juntamente com os códigos desenvolvidos, no endereço indicado na Seção 6.1.

7.1.2 Definição dos Testes

Os testes foram planejados de modo a atender da melhor maneira possível o objetivo 6 indicado na Seção 1.2: *"Comparar os desempenhos dos benchmarks compilados com o método implementado, com os desempenhos desses mesmos benchmarks compilados com os métodos de alocação e escalonamento existentes no LLVM"*. Esse objetivo foi estratificado nas seguintes diretrizes, que nortearam o planejamento:

- verificar o efeito geral da abordagem de Pinter no desempenho dos programas;
- verificar o efeito da abordagem no derramamento de variáveis para a memória;
- verificar o esforço necessário para a compilação dos programas utilizando a abordagem de Pinter.

A partir dessas diretrizes, a seguinte sequência de testes foi definida:

T1: executar os *benchmarks* compilados com os métodos de alocação de registradores existentes no LLVM e determinar qual desses métodos gera códigos com melhor desempenho.

T2: executar os *benchmarks* compilados com as diferentes heurísticas implementadas para a abordagem de Pinter e determinar qual delas gera códigos com melhor desempenho.

T3: comparar os resultados obtidos nos testes T1 e T2, para determinar se a implementação realizada gera resultados aceitáveis.

T4: executar os *benchmarks* compilados com o Microsoft Visual Studio e comparar os resultados de desempenho obtidos com os resultados dos testes T1 e T2.

T5: comparar os métodos de alocação de registradores existentes no LLVM com as diferentes heurísticas utilizadas na implementação da abordagem de Pinter considerando os derramamentos de variáveis para a memória.

T6: comparar os esforços de compilação dos métodos de alocação de registradores existentes no LLVM com os esforços necessários à compilação das diferentes heurísticas definidas para a abordagem de Pinter.

T7: avaliar o efeito do momento da execução da tarefa de escalonamento de instruções sobre o desempenho das diversas compilações dos *benchmarks*.

7.1.2.1 Testes T1 a T4

Como o problema da interdependência é eminentemente teórico, seria razoável propor que a abordagem implementada fosse avaliada a partir de uma análise minuciosa do código por ela produzido. Uma análise capaz de verificar se as interferências entre as tarefas de alocação de registradores e escalonamento de instruções foram minimizadas, se o código apresentou um melhor sequenciamento de instruções, ou utilizou de forma mais adequada os registradores.

Apesar dessa proposição ser aparentemente mais estimulante, do ponto de vista teórico, pareceu-nos ser também mais suscetível a subjetividades, pois, no mínimo, seria difícil encontrar parâmetros de avaliação adequados para validar quaisquer resultados que fossem encontrados. Na análise do mesmo código, por exemplo, poderíamos encontrar situações conflitantes, uma apontando na direção de um código de melhor qualidade, outra apontando na direção oposta, e não teríamos como afirmar qual delas prevaleceria sobre a outra, pois, de fato, não há uma solução para o problema, nem uma referência que poderia ser utilizada para justificar teoricamente qualquer conclusão.

De forma contrária à consideração feita sobre uma análise teórica, os testes propostos apontam na direção de uma avaliação sistêmica e empírica. Tal proposta parece-nos ser igualmente justificável, pois o problema também possui um forte viés pragmático: afinal, o objetivo último da abordagem proposta por Pinter, e esse também é o objetivo de todos os outros pesquisadores que se dedicaram sobre o tema, é fazer com que as interferências entre as tarefas de alocação e escalonamento sejam minimizadas a ponto de se ter um código final com melhor desempenho.

Assim, as avaliações empíricas propostas, ainda que não provem formalmente, no mínimo são capazes de indicar de maneira aceitável a qualidade da implementação da abordagem no tratamento do problema, pois seus resultados podem ser mensurados de modo mais preciso e confiável.

Para os testes T1, T2 e T4, a dinâmica adotada foi a mesma. Todos os *benchmarks* foram executados no ambiente SPEC utilizando os três conjuntos de dados disponíveis, *test*, *train* e *ref*. Os resultados relacionados aos conjuntos *test* e *train* foram obtidos por meio de execuções individuais. O resultado final da execução com o conjunto de dados *ref* foi a média aritmética das três medições fornecidas pela *execução completa* dos *benchmarks*, descrita na Seção 5.2. Os resultados fornecidos pelas execuções utilizando os conjuntos de dados *test* e *train* apresentaram as mesmas tendências daqueles que foram obtidos com o conjunto de dados *ref*. Assim, por serem redundantes, a Seção 7.2 considera apenas os resultados relacionados ao conjunto de dados *ref*.

Para o teste T1, todos os quatro métodos de alocação de registradores existentes

no LLVM foram considerados: *basic*, *greedy*, *PBQP* e *fast*.

Especificamente para o teste T2, o grafo de interferência paralelizável foi utilizado pelo alocador de registradores desenvolvido utilizando as heurísticas *H2* e *HB* apresentadas nas seções 6.1.1.6 e 6.1.1.8 respectivamente. Nos testes feitos durante o desenvolvimento, a heurística *H1* apresentou um desempenho bastante inferior à heurística *H2* em termos do tempo necessário para completar a tarefa de alocação e, por essa razão, ela não foi considerada nos testes. Na Seção 7.2, os métodos associados às heurísticas *H2* e *HB* estão identificados por *PinterH2* e *PinterHB* respectivamente.

Ainda no Teste T2, a utilização do parâmetro *pinter-pgraphisigraph*, descrito na Seção 6.1.2, também foi considerada. Esse parâmetro faz com que o alocador de registradores trabalhe com o grafo de interferência tradicional, sem a inserção de quaisquer arestas de falsas dependências. Assim, foi possível comparar o processo de alocação utilizando o grafo de interferência tradicional com os processos de alocação utilizando o grafo de interferência paralelizável. A heurística padrão *H2* foi a escolhida para ser utilizada com esse parâmetro. Em função disso, nas tabelas e gráficos da Seção 7.2, o método associado à utilização desse parâmetro está identificado por *ColorH2*.

O Teste T4 foi incluído no planejamento em função do Microsoft Visual Studio ser o compilador indicado pela Microsoft para a plataforma Windows. Nesse teste, todos os *benchmarks* e *pseudo-benchmarks* indicados na Seção 7.1.2 foram compilados com o Visual Studio, exceto o *benchmark 462.libquantum*. Isso porque o Visual Studio utiliza a biblioteca UCRT (*Universal C Run-Time*), que implementa o padrão ISO/IEC 9899:1999, também conhecido como C99, que é incompatível com alguns dos tipos de dados utilizados por esse *benchmark*, definidos em <Complex.h> [Corporation, 2017a].

7.1.2.2 Teste T5

A abordagem de Pinter tende a favorecer a tarefa de escalonamento de instruções. Ao inserir arestas no grafo de interferência, para evitar o surgimento de falsas dependências no código, a tendência é que um número maior de registradores seja utilizado e, portanto, a possibilidade de haver derramamentos de valores para a memória tende a crescer. Todavia, esse não é um indicador confiável para avaliar a implementação da abordagem, pois, uma vez que o número de derramamentos para a memória será tomado comparativamente, o tratamento dado a essa questão pelos métodos com os quais a implementação será comparada podem influir bastante no resultado.

Por outro lado, esse pode ser um bom indicador para justificar o desempenho de qualquer um dos métodos considerados nos testes, uma vez que é sabido que os derramamentos, em geral, prejudicam o desempenho. A relevância deste ponto aumenta,

quando consideramos a arquitetura utilizada nos testes. Trata-se de uma arquitetura CISC, mas que utiliza tecnologia superescalar, incluindo a iniciação fora de ordem de instruções. Considerando os resultados apresentados em Valluri & Govindarajan [1999], essa arquitetura deveria se beneficiar de uma melhor alocação dos registradores e, portanto, de um menor número de derramamentos. Assim, os melhores resultados de desempenho, provavelmente, deveriam estar atrelados aos métodos capazes de minimizar essas ocorrências.

Este teste, assim como os testes T6 e T7, não considera o Microsoft Visual Studio. Para esse teste, os dados relativos a esse compilador não podem ser obtidos. Para os dois testes seguintes, não há como realizar nesse compilador as tarefas necessárias.

7.1.2.3 Teste T6

Trata-se de um teste interessante para avaliar a viabilidade de utilização dos métodos, ou determinar em que circunstâncias eles podem ser utilizados. Tais testes, entretanto, não possuem relação direta com o problema da interdependência em si. Neste teste, apenas os tempos envolvidos nos processos de tradução dos códigos fonte para os códigos objeto foram considerados. O tempo do processo de amarração, realizado pelo Visual Studio, não foi considerado, por ter sido praticamente insignificante em todos os casos.

7.1.2.4 Teste T7

A abordagem de Pinter baseia-se no escalonamento *postpass*, por isso, em todos os testes anteriores, a tarefa de escalonamento é executada após a tarefa de alocação. Todavia, a arquitetura utilizada nos testes parece dispensar o escalonamento de instruções realizado pelo compilador, uma vez que, como foi dito, trata-se de uma arquitetura capaz de escalonar instruções dinamicamente, iniciando-as fora de ordem. Caso seja realmente assim, este teste será interessante, pois os resultados obtidos por meio dele evidenciarão e comprovarão essa característica.

Testes com processadores ARM, que utilizam arquitetura RISC, foram definidos, mas devido à indisponibilidade do *hardware* não puderam ser feitos. Teoricamente, testes com quaisquer modelos de quaisquer famílias de processadores ARM, tais como ARM7, ARM9, ARM11, Cortex-M, Cortex-R e Cortex-A, com exceção do modelo Cortex-A15 MPCore, que possui iniciação de instruções fora de ordem [Infocenter, 2017], deveriam apresentar resultados diferentes dos que foram esperados e obtidos neste teste.

7.2 Resultados

Os resultados de cada um dos sete testes definidos são apresentados separadamente. As tabelas apresentam os dados relativos aos *benchmarks* e aos *pseudo-benchmarks*. Os gráficos, por praticidade e tendo em vista que nenhuma mudança de resultado seria observada, consideram somente os *benchmarks*.

Todos os resultados apresentados nesta seção referem-se às execuções dos *benchmarks* com o conjunto de dados *ref*. Os executáveis utilizados nos testes T1 a T6, foram gerados com escalonamento de instruções *postpass*.

A Tabela 7.2 apresenta os tempos de execução dos *benchmarks* compilados com cada um dos quatro métodos existentes no LLVM e também com as três abordagens implementadas e consideradas nos testes: *PinterH2*, *ColorH2* e *PinterHB*. Todos os valores existentes nessa tabela representam tempos em segundos e foram obtidos com os testes T1 e T2.

Benchmark	**Basic**	**Greedy**	**PBQP**	**Fast**	**PinterH2**	**ColorH2**	**PinterHB**
401.bzip2	947,31	864,54	972,74	1563,37	785,91	972,56	584,40
429.mcf	320,00	302,77	317,15	344,08	264,04	317,12	212,21
433.milc	519,56	518,12	518,37	457,36	468,72	519,20	403,90
445.gobmk	841,39	761,61	812,42	822,74	642,33	827,98	502,61
458.sjeng	809,04	787,84	811,23	1237,03	679,17	832,27	505,67
462.libquantum	584,65	560,87	579,92	1049,48	495,60	580,33	394,46
464.h264ref	1131,31	1107,74	1293,37	948,34	1107,08	1321,31	757,88
470.lbm	494,49	467,56	456,50	355,28	410,85	502,35	303,93
482.sphinx3	973,09	959,46	972,13	1110,35	797,55	983,61	621,10
998.specrand	0,39	0,39	0,41	0,31	0,37	0,42	0,30
999.specrand	0,39	0,40	0,40	0,28	0,37	0,40	0,29
473.astar	679,29	626,33	670,25	706,77	527,26	679,69	420,16

Tabela 7.2. Tempos das execuções com a abordagem *postpass* (segundos).

Teste T1

Com o Teste T1 buscamos identificar o método de alocação de registradores do LLVM capaz de gerar executáveis com melhor desempenho.

A Figura 7.2 apresenta de forma gráfica os dados da Tabela 7.2 que são pertinentes a este teste. Os dois gráficos dessa figura apresentam as mesmas informações, mas de formas diferentes. Pelos dados da tabela, é possível verificar que todas as compilações com melhor desempenho estão associadas aos métodos *Greedy* e *Fast*. O primeiro deles está associado a sete dessas compilações. O segundo está relacionado a três delas. Interessante notar que, em comparação com os demais, o método *Fast* comporta-se de modo bastante irregular.

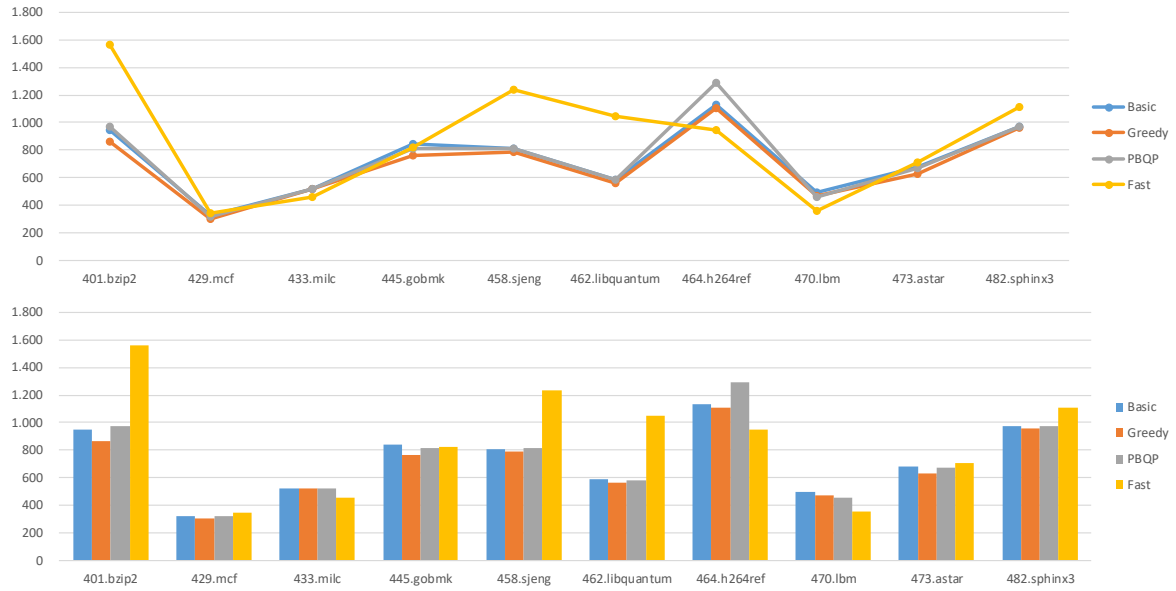


Figura 7.2. Resultados dos métodos existentes no LLVM.

Teste T2

Este teste procurou identificar a heurística utilizada com a abordagem de Pinter que gerou os executáveis com melhor desempenho.

A Figura 7.3 mostra os dois gráficos associados aos dados da Tabela 7.2 que são pertinentes a este teste. É possível verificar que a abordagem de Pinter utilizando a heurística H8 forneceu melhores resultados para todos os *benchmarks* testados. Além disso, é possível notar também que todas as heurísticas implementadas comportaram-se da mesma maneira.

A heurística H2 leva em consideração aspectos práticos do LLVM. Ela retira do grafo de interferência paralelizável, em primeiro lugar, os vértices cujo derramamento para a memória não pode ser feito, isto é, aqueles vértices que se forem selecionados para derramamento gerarão erro durante o processo de alocação de registradores. Trata-se, portanto, de uma heurística que garante maior robustez ao processo, mas que reduz o seu desempenho.

O método *ColorH2* também utiliza a heurística H2, mas, ao invés de trabalhar com o grafo de interferência paralelizável para realizar a alocação de registradores, ele utiliza o grafo de interferência tradicional, no qual nenhuma aresta de falsas dependências foi incluída. Este método equivale a um método de alocação de registradores tradicional, baseado na coloração do grafo de interferência.

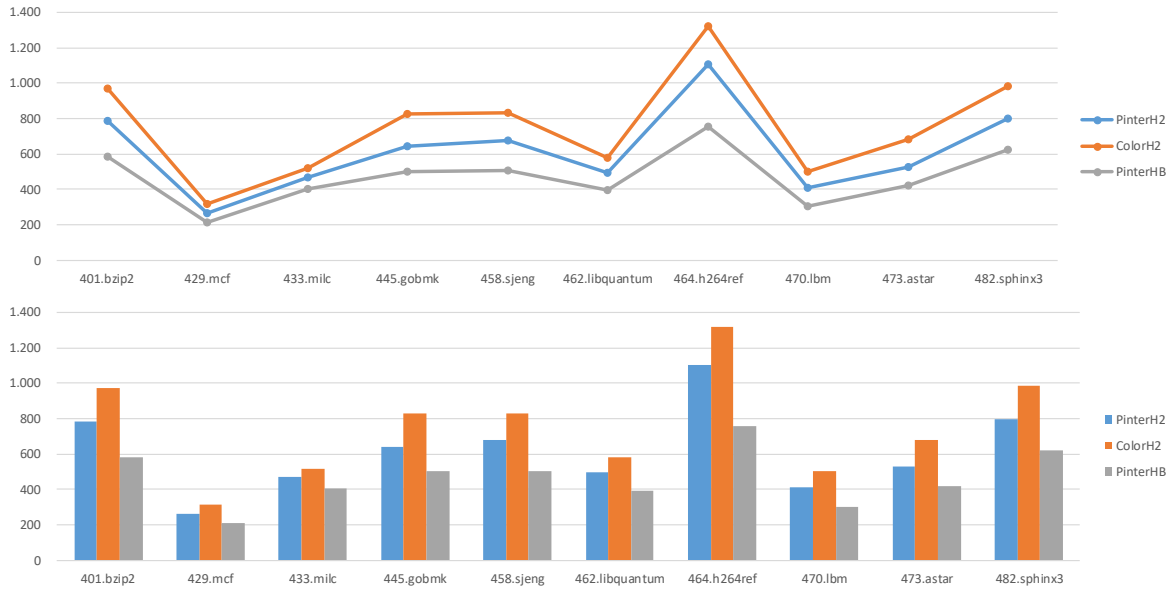


Figura 7.3. Resultados da abordagem de Pinter com as abordagens implementadas.

Teste T3

No Teste T3, comparamos os resultados dos dois testes anteriores, com o objetivo de determinar possíveis ganhos da abordagem implementada sobre os métodos de alocação de registradores do LLVM.

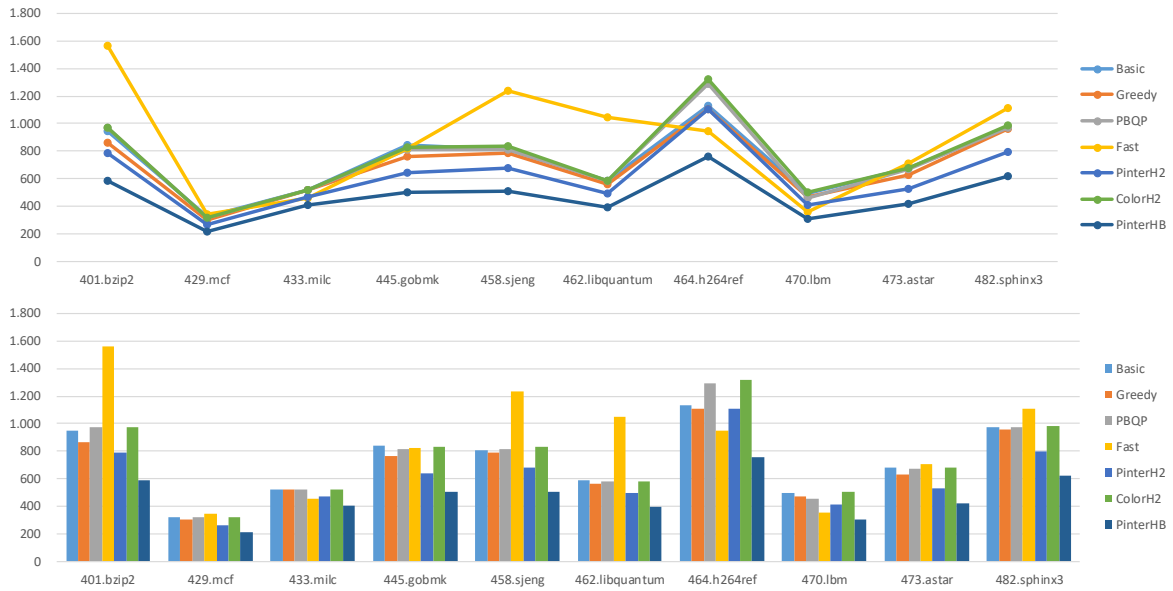


Figura 7.4. Desempenho dos *benchmarks* compilados com os métodos do LLVM e com as abordagens implementadas.

A Figura 7.4 exibe os desempenhos dos *benchmarks* compilados com cada um dos métodos do LLVM em relação a cada uma das heurísticas utilizadas com a abordagem de Pinter.

Figura 7.5. Comparação entre cada um dos métodos do LLVM e as abordagens implementadas. Comparação das abordagens com os métodos: (a) *Basic*; (b) *Greedy*; (c) *PBQP* e (d) *Fast*.

Por sua vez, os gráficos (a), (b), (c) e (d) da Figura 7.5 estratificam os resultados

da Figura 7.4. Eles comparam os desempenhos das heurísticas desenvolvidas (seções 6.1.1.6 e 6.1.1.8) com os desempenhos dos métodos *Basic*, *Greedy*, *PBQP* e *Fast* do LLVM respectivamente, mostrando que esses métodos possuem desempenho inferior.

Conforme os dados das tabelas 7.3 e 7.4, as abordagens de Pinter utilizando as heurísticas HB e H2, principalmente a primeira delas, mostraram-se bem mais efetivas do que os dois melhores métodos do LLVM identificados no Teste T1, *Greedy* e *Fast*.

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
PinterHB	22,05	35,82	31,86	4,06
PinterH2	0,06	16,88	11,74	4,86
ColorH2	-19,28	-0,21	-7,30	5,49

Tabela 7.3. Ganhos da abordagem de Pinter sobre o método *Greedy* do LLVM.

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
PinterHB	11,69	62,62	39,22	18,99
PinterH2	-16,74	52,78	21,15	25,41
ColorH2	-41,40	44,70	4,34	29,74

Tabela 7.4. Ganhos da abordagem de Pinter sobre o método *Fast* do LLVM.

Teste T4

Neste teste, os resultados dos testes T1 e T2 foram comparados com os desempenhos dos *benchmarks* compilados com o Visual Studio. A Tabela 7.5 mostra os tempos de execução associados a cada uma dessas compilações. Como explicado na Seção 7.1.2, o *benchmark 462.libquantum* não pôde ser compilado com esse compilador.

Benchmark	**Tempo Visual Studio**
401.bzip2	697,16
429.mcf	262,54
433.milc	442,97
445.gobmk	577,73
458.sjeng	636,08
462.libquantum	-
464.h264ref	746,22
470.lbm	371,38
482.sphinx3	782,74
998.specrand	0,42
999.specrand	0,38
473.astar	451,41

Tabela 7.5. Tempos das execuções com o Visual Studio (segundos).

Os gráficos das figuras 7.6 e 7.7 apresentam, respectivamente, a comparação dos resultados do Teste T1 (métodos do LLVM) e dos resultados do Teste T2 (heurísticas implementadas para a abordagem de Pinter) com os desempenhos dos *benchmarks* compilados com o Visual Studio.

Figura 7.6. Comparação dos métodos do LLVM com o Visual Studio.

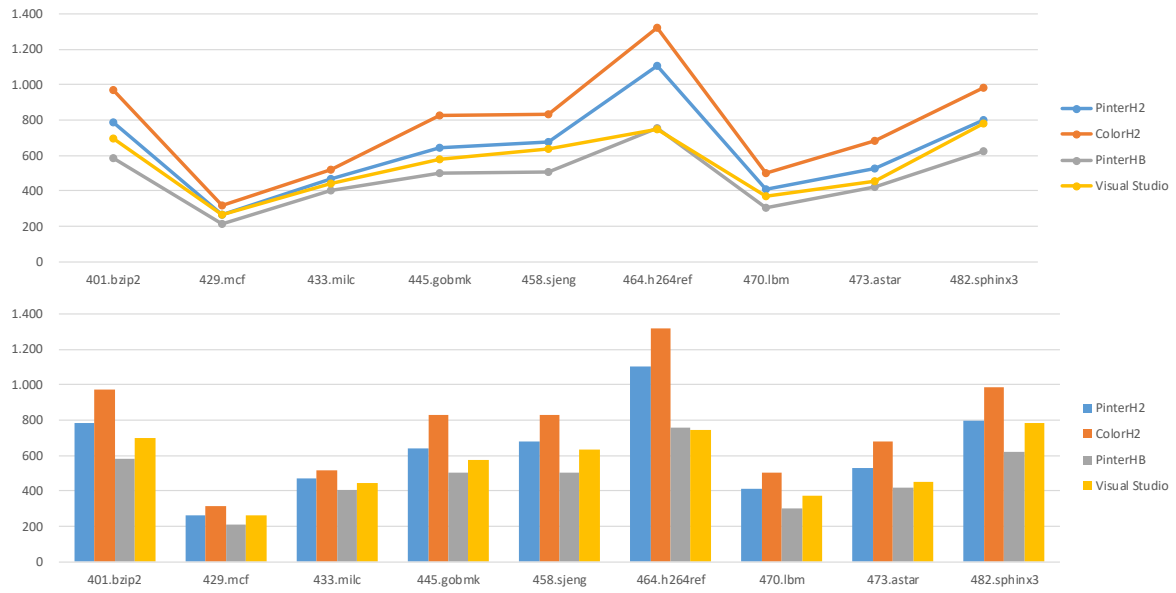


Figura 7.7. Comparação das heurísticas utilizadas com a abordagem de Pinter com o Visual Studio.

Esses gráficos mostram que, em relação aos métodos do LLVM, o Visual Stu-

dio forneceu códigos com desempenho superior. Por outro lado, a implementação da abordagem de Pinter utilizando a heurística HB apresentou resultados melhores do que o Visual Studio em quase todos os casos. De fato, apenas o *benchmark 464.h264ref* apresentou um desempenho um pouco inferior, cerca de 1,6% menor. Os ganhos da abordagem de Pinter sobre o Visual Studio podem ser vistos na Tabela 7.6. Os ganhos das compilações feitas com o Visual Studio sobre as compilações feitas com os dois melhores métodos do LLVM, *Greedy* e *Fast*, podem ser vistos na Tabela 7.7.

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
PinterHB	-1,56	20,65	13,54	7,53
PinterH2	-48,36	-0,57	-12,75	14,32
ColorH2	-77,07	-17,21	-37,80	18,21

Tabela 7.6. Ganhos da abordagem de Pinter sobre o Visual Studio.

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
Greedy	13,29	32,64	21,12	6,20
Fast	-4,53	55,41	27,00	19,29

Tabela 7.7. Ganhos do Visual Studio sobre os métodos do LLVM.

Teste T5

O Teste T5 avaliou os métodos sob o ponto de vista dos derramamentos de valores para a memória. As tabelas 7.8 e 7.9 exibem os totais de derramamentos produzidos. Elas apresentam as mesmas informações, mas de modos diferentes.

Benchmark	**Basic**	**Greedy**	**PBQP**	**Fast**	**PinterH2**	**ColorH2**	**PinterHB**
401.bzip2	445	802	529	17558	569	531	443
429.mcf	84	101	86	829	94	89	84
433.milc	995	1392	1012	7322	1039	1023	978
445.gobmk	3625	4855	3882	46225	3923	3824	3539
458.sjeng	371	537	400	8750	413	397	358
462.libquantum	437	632	429	2510	472	465	428
464.h264ref	5615	8496	6557	34031	6662	6411	5620
470.lbm	143	239	146	460	149	149	145
482.sphinx3	1060	1534	1130	11412	1173	1142	1072
998.specrand	2	2	2	24	2	2	2
999.specrand	2	2	2	24	2	2	2
473.astar	460	505	481	3280	495	478	465

Tabela 7.8. Totais absolutos de derramamentos para a memória.

Benchmark	Basic	Greedy	PBQP	Fast	PinterH2	ColorH2	PinterHB	Total
401.bzip2	2,13	3,84	2,53	84,10	2,73	2,54	2,12	100,00
429.mcf	6,14	7,39	6,29	60,64	6,88	6,51	6,14	100,00
433.milc	7,23	10,12	7,35	53,21	7,55	7,43	7,11	100,00
445.gobmk	5,19	6,95	5,56	66,16	5,61	5,47	5,06	100,00
458.sjeng	3,30	4,78	3,56	77,94	3,68	3,54	3,19	100,00
462.libquantum	8,13	11,76	7,98	46,72	8,78	8,65	7,97	100,00
464.h264ref	7,65	11,58	8,93	46,37	9,08	8,74	7,66	100,00
470.lbm	9,99	16,70	10,20	32,15	10,41	10,41	10,13	100,00
482.sphinx3	5,72	8,28	6,10	61,61	6,33	6,17	5,79	100,00
998.specrand	5,56	5,56	5,56	66,67	5,56	5,56	5,56	100,00
999.specrand	5,56	5,56	5,56	66,67	5,56	5,56	5,56	100,00
473.astar	7,46	8,19	7,80	53,21	8,03	7,75	7,54	100,00

Tabela 7.9. Totais relativos de derramamentos para a memória considerando o total absoluto associado a cada um dos *benchmarks* (porcentagens).

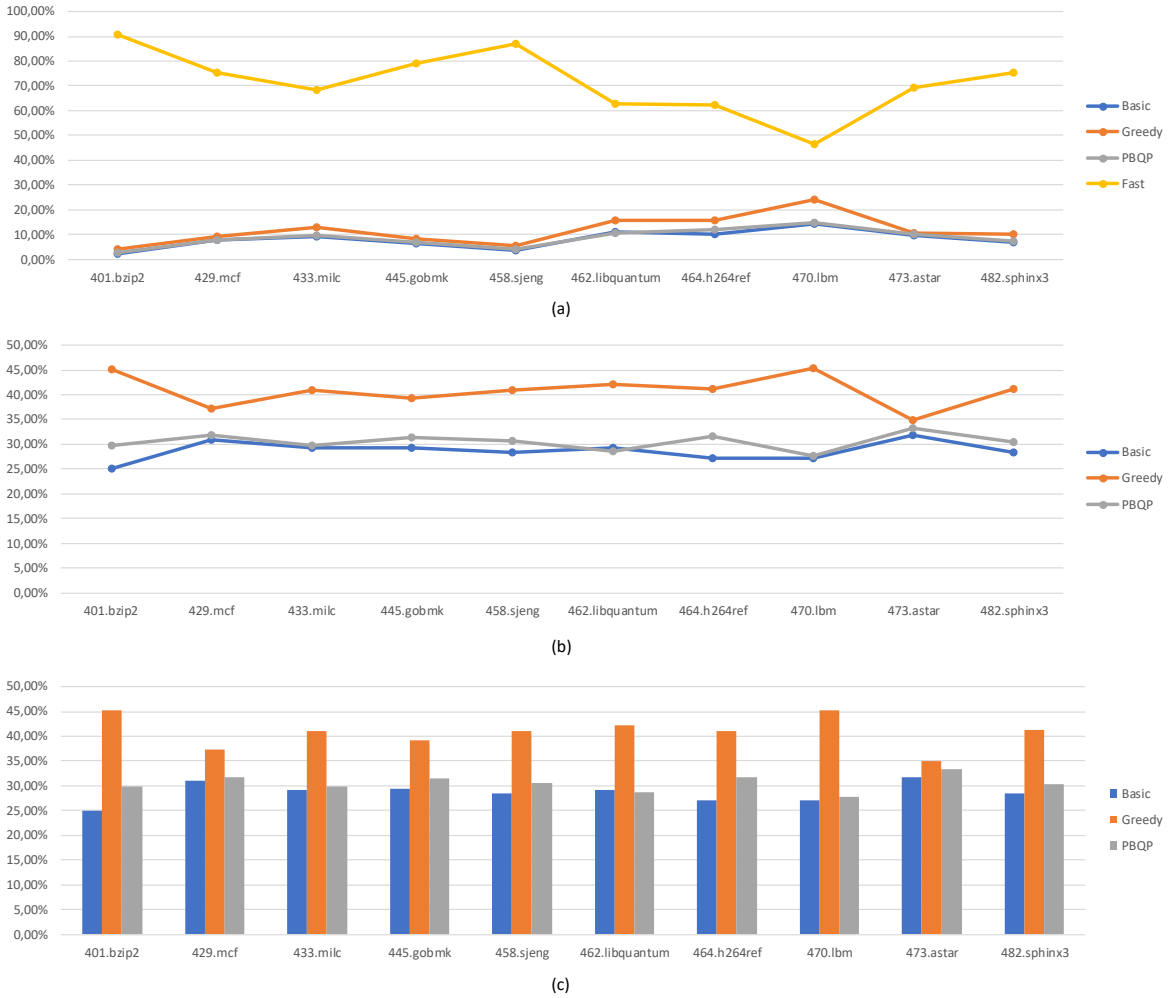


Figura 7.8. Derramamentos produzidos pelos métodos do LLVM: (a) considerando o método *Fast*; (b) e (c) desconsiderando o método *Fast*.

A primeira apresenta o total absoluto de derramamentos produzidos por cada um dos métodos para cada um dos *benchmarks*. A segunda, a fim de facilitar a comparação dos valores, mostra os totais relativos de derramamento, correspondentes a cada um dos métodos, considerando o total absoluto associado a cada um dos *benchmarks*.

Não foi possível apurar de forma clara e precisa o número de derramamentos relacionados ao método *Fast* do LLVM. Na realidade, o código relacionado a este método possibilita a extração de três estatísticas: *stores*, *loads* e *copies*. Todas elas relacionam-se com o número de derramamentos feitos, mas nenhuma delas reflete com precisão esse número. Assim, assumimos que a primeira delas indicaria o número de derramamentos para a memória. Essa é de fato uma escolha plausível, mas também imprecisa e um tanto quanto arbitrária.

Os gráficos (a), (b) e (c) da Figura 7.8 relacionam-se ao número de derramamentos dos métodos do LLVM. O primeiro deles considera o método *Fast* e os dois últimos, pela razão exposta anteriormente, desconsideram esse método. Pelos gráficos, é possível ver que o método *Greedy* está associado ao maior número de derramamentos em todos os casos, enquanto o método *Basic* produz o menor número de derramamentos em quase todas as situações.

Os gráficos da Figura 7.9 mostram os derramamentos produzidos pelas heurísticas implementadas. A heurística HB produz o menor número de derramamentos para todos os *benchmarks*, seguida pela heurística H2 e pelo método *ColorH2*.

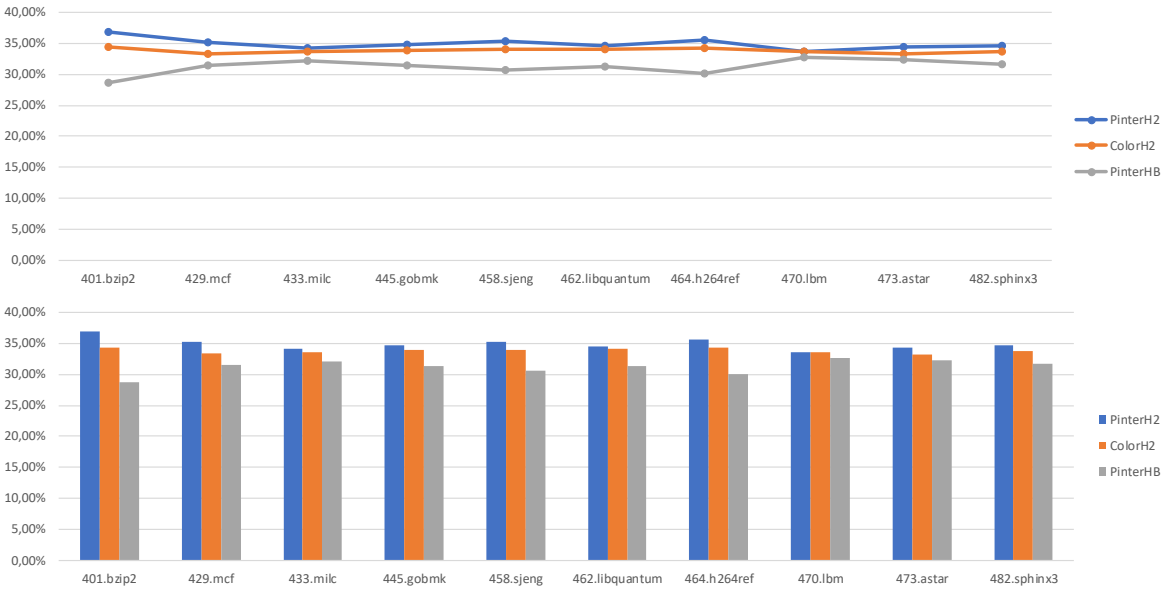


Figura 7.9. Comparação das heurísticas utilizadas com a abordagem de Pinter com o Visual Studio.

As informações fornecidas por esses gráficos mostram que o desempenho de cada

um dos métodos não pode ser totalmente explicado levando-se em conta apenas o número de derramamentos para a memória. Considerando os métodos existentes no LLVM, o método *Greedy* é o que produz o maior número de derramamentos, mas também é o que produz o maior número de compilações com melhor desempenho. Por outro lado, considerando as heurísticas implementadas para a abordagem de Pinter, a heurística HB está associada aos melhores desempenhos e também aos menores números de derramamentos. Todavia, como mostram os gráficos da Figura 7.10, que comparam os métodos *Basic* e a heurística HB, por serem eles os que apresentaram os menores totais de derramamentos para a memória, a heurística HB produz um número igual ou maior de derramamentos em 50% dos casos testados.

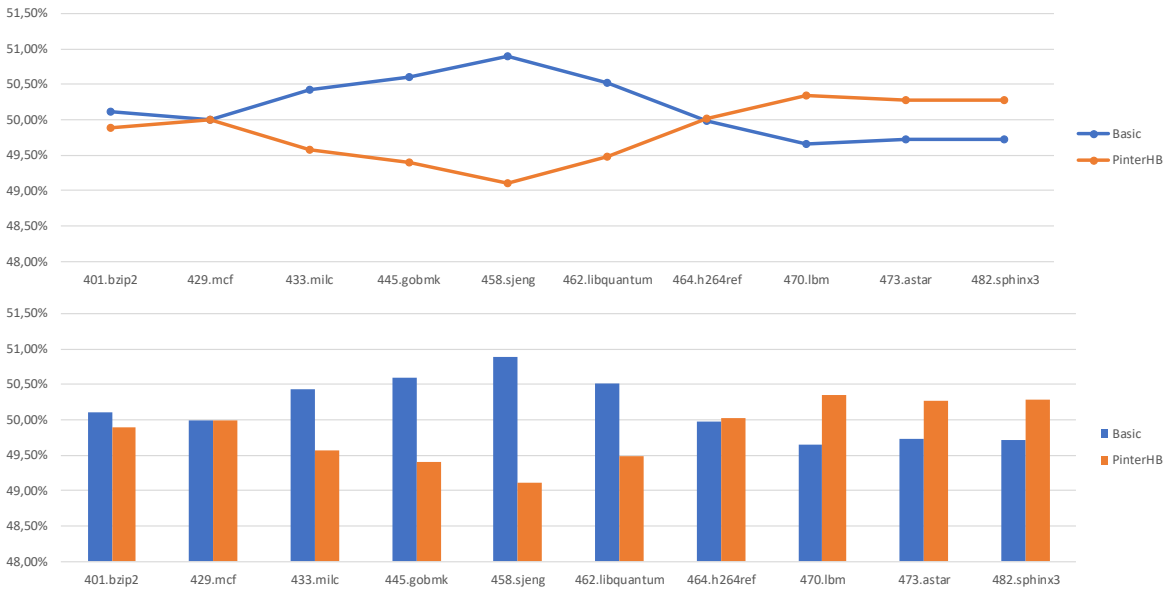


Figura 7.10. Comparação das heurísticas utilizadas com a abordagem de Pinter com o Visual Studio.

Teste T6

Este teste mede os tempos necessários à compilação dos *benchmarks* com os diferentes métodos. Ele não está associado a nenhum fator que possa ser diretamente relacionado ao problema da interdependência, contudo ele foi incluído na sequência de testes por ser útil para indicar as situações nas quais seria viável utilizar a implementação feita.

As tabelas 7.10 e 7.11 apresentam, respectivamente, os tempos necessários à geração dos *bitcodes* com CLang e à criação dos códigos objetos com LLC. Finalmente, a Tabela 7.12 apresenta o tempo total de compilação de cada *benchmark* para cada um

Benchmark	Tempo CLang
401.bzip2	4,87
429.mcf	0,70
433.milc	2,82
445.gobmk	14,18
458.sjeng	2,15
462.libquantum	0,73
464.h264ref	11,88
470.lbm	0,23
482.sphinx3	4,20
998.specrand	0,05
999.specrand	0,05
473.astar	0,98

Tabela 7.10. Tempos para a geração dos *bitcodes* (minutos).

	Tempo LLC		
Benchmark	PinterH2	ColorH2	PinterHB
401.bzip2	3,07	1,35	1,77
429.mcf	0,55	0,52	0,53
433.milc	3,82	3,50	3,73
445.gobmk	9,45	6,95	7,97
458.sjeng	1,65	1,35	1,53
462.libquantum	0,98	0,87	0,95
464.h264ref	27,95	11,90	16,22
470.lbm	0,23	0,15	0,22
482.sphinx3	3,05	2,65	2,85
998.specrand	0,08	0,08	0,08
999.specrand	0,08	0,08	0,08
473.astar	0,80	0,70	0,75

Tabela 7.11. Tempos para a geração dos códigos objeto (minutos).

Benchmark	PinterH2	ColorH2	PinterHB	Total
401.bzip2	7,93	6,22	6,63	20,78
429.mcf	1,25	1,22	1,23	3,70
433.milc	6,63	6,32	6,55	19,50
445.gobmk	23,63	21,13	22,15	66,92
458.sjeng	3,80	3,50	3,68	10,98
462.libquantum	1,72	1,60	1,68	5,00
464.h264ref	39,83	23,78	28,10	91,72
470.lbm	0,47	0,38	0,45	1,30
482.sphinx3	7,25	6,85	7,05	21,15
998.specrand	0,13	0,13	0,13	0,40
999.specrand	0,13	0,13	0,13	0,40
473.astar	1,78	1,68	1,73	5,20
Total	94,57	72,95	79,53	247,05

Tabela 7.12. Tempos totais de compilação (minutos).

dos métodos implementados. Os gráficos (a), (b) e (c) da Figura 7.11 exibem os dados de cada uma dessas três tabelas.

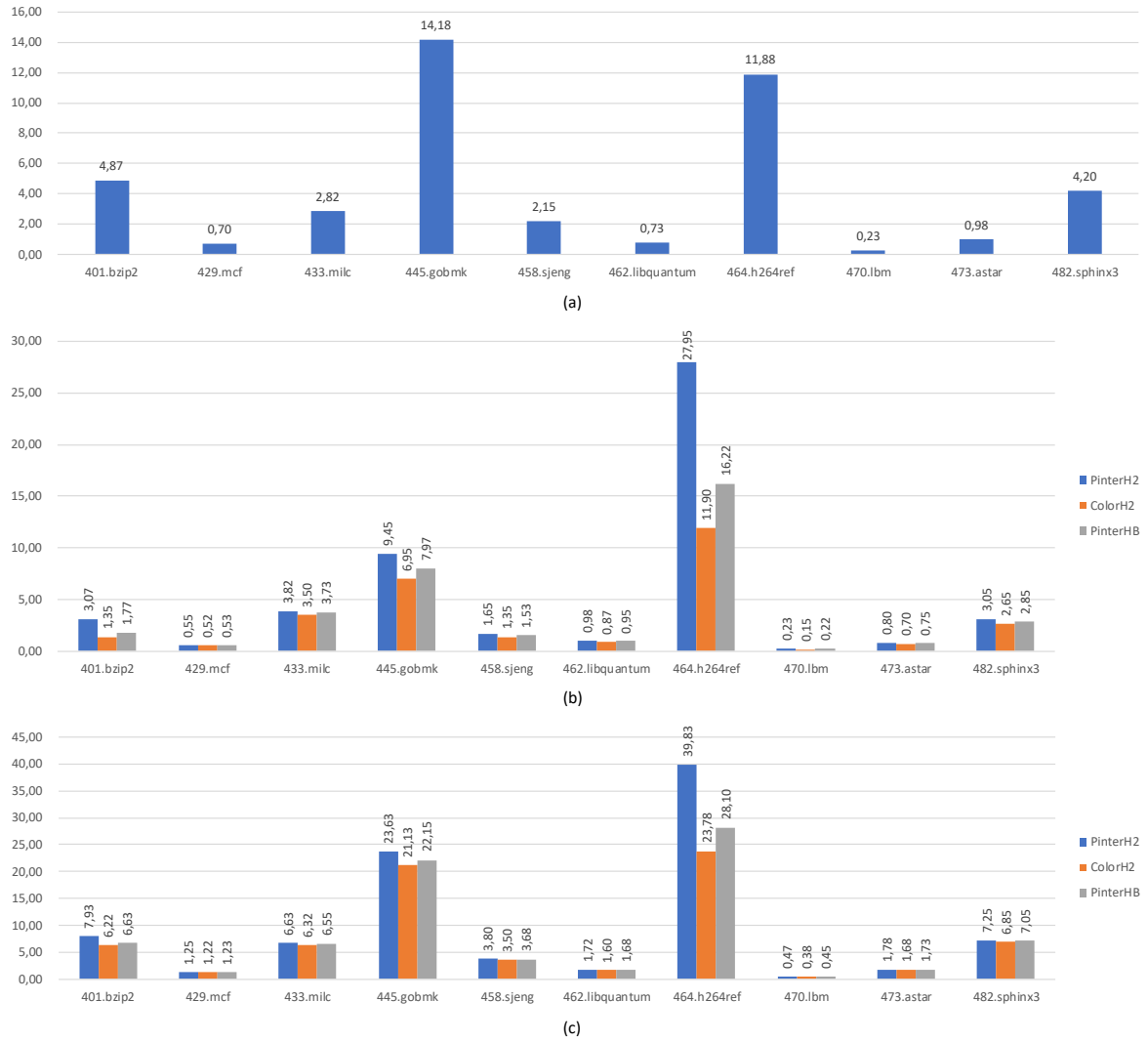


Figura 7.11. Tempos de compilação associados à abordagem implementada: (a) tempo relacionado à geração dos *bitcodes* pelo *frontend* CLang; (b) tempo para a geração dos códigos objeto pelo compilador LLC; (c) tempo total de compilação.

A Tabela 7.13 e o gráfico da Figura 7.12 mostram os tempos de compilação de cada um dos *benchmark* para cada um dos métodos existentes no LLVM. O gráfico da Figura 7.13 apresenta os tempos de compilação associados a todos os métodos, dando uma ideia da diferença existente entre eles.

É possível ver que os três métodos implementados e testados demoraram em conjunto mais de 4 horas para compilarem todos os *benchmarks*, enquanto os quatro métodos existentes no LLVM, mesmo sendo em maior em número, precisaram ao todo de 24 minutos somente, um tempo 10 vezes menor aproximadamente.

Assim, este teste revela que a implementação feita é inviável para ambientes de desenvolvimento e teste de software. Contudo, ela poderia ser utilizada para a produção

final de programas executáveis que integram sistemas embarcados, ou que se destinam aos usuários finais de uma determinada aplicação.

Benchmark	**Basic**	**Greedy**	**PBQP**	**Fast**	**Total**
401.bzip2	0,30	0,32	0,37	0,08	1,07
429.mcf	0,07	0,07	0,07	0,03	0,23
433.milc	0,52	0,55	0,55	0,23	1,85
445.gobmk	2,95	3,05	3,12	1,00	10,12
458.sjeng	0,37	0,37	0,40	0,13	1,27
462.libquantum	0,15	0,17	0,17	0,07	0,55
464.h264ref	1,83	1,92	2,50	0,52	6,77
470.lbm	0,03	0,03	0,03	0,02	0,12
482.sphinx3	0,53	0,55	0,58	0,20	1,87
998.specrand	0,02	0,00	0,00	0,02	0,03
999.specrand	0,00	0,00	0,02	0,00	0,02
473.astar	0,15	0,15	0,15	0,08	0,53
Total	6,92	7,17	7,95	2,38	24,42

Tabela 7.13. Tempos de compilação com os métodos do LLVM (minutos).

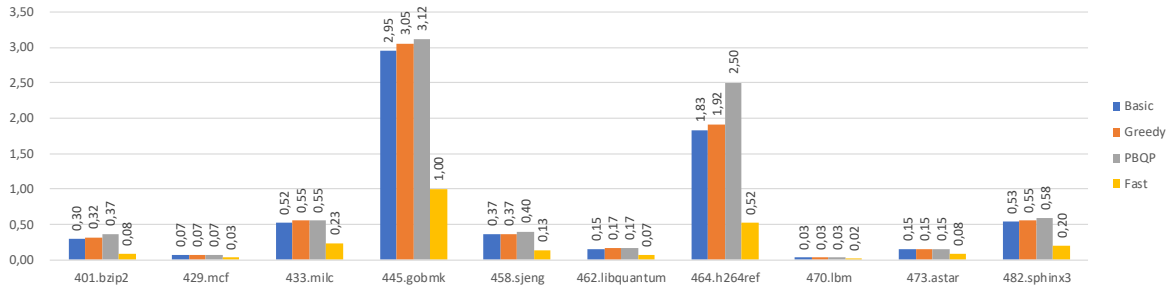


Figura 7.12. Tempos de compilação relacionados aos métodos do LLVM.

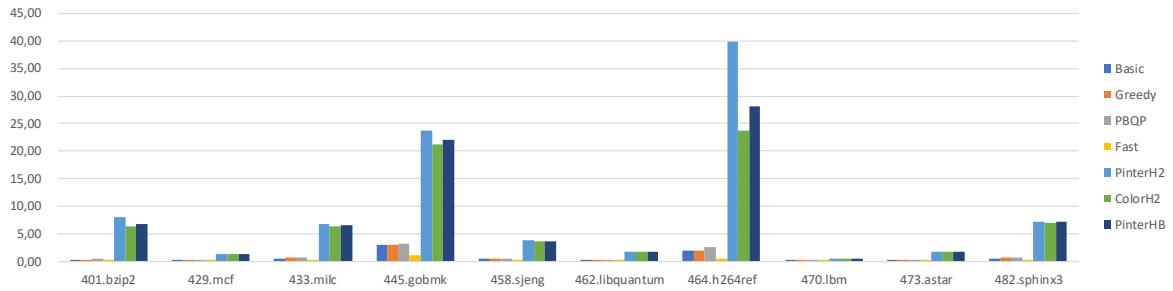


Figura 7.13. Tempos de compilação dos diferentes métodos.

Teste T7

O Teste T7 está relacionado ao momento de execução da tarefa de escalonamento de instruções. Para este teste, todos os *benchmarks* foram compilados com o escalona-

mento *prepass* e executados com o mesmo conjunto de dados *ref* utilizado nos testes anteriores. Os tempos de execução dos programas compilados com escalonamento *postpass* foram apresentados na Tabela 7.2. Os tempos de execução dos programas compilados com o escalonamento *prepass* podem ser vistos na Tabela 7.14.

Benchmark	**Basic**	**Greedy**	**PBQP**	**Fast**	**PinterH2**	**ColorH2**	**PinterHB**
401.bzip2	945,92	874,90	983,99	1577,44	808,23	1001,61	590,78
429.mcf	318,16	304,60	312,55	342,83	264,68	320,43	210,55
433.milc	516,95	519,59	519,41	456,75	467,99	519,13	402,25
445.gobmk	838,46	766,43	812,25	822,70	627,87	830,61	501,79
458.sjeng	809,61	796,30	805,98	1234,63	684,05	828,31	501,63
462.libquantum	583,76	558,81	569,61	1052,85	489,23	576,14	396,87
464.h264ref	1129,59	1106,73	1287,41	944,01	1081,34	1301,23	749,27
470.lbm	498,55	466,59	469,51	347,89	429,42	508,37	297,38
482.sphinx3	974,04	960,13	978,97	1104,48	798,28	991,06	624,22
998.specrand	0,40	0,39	0,39	0,29	0,41	0,43	0,29
999.specrand	0,41	0,41	0,43	0,29	0,37	0,39	0,29
473.astar	679,88	622,87	668,28	702,05	546,81	678,34	422,68

Tabela 7.14. Tempos das execuções com a abordagem *prepass* (segundos).

Método	**Média (%)**	**Desvio Padrão**
Basic	49,85	0,00399
Greedy	49,84	0,00293
PBQP	49,97	0,00688
Fast	50,17	0,00547
PinterH2	49,66	0,00901
ColorH2	49,96	0,00370
PinterHB	50,13	0,00337

Tabela 7.15. Médias de execução *postpass* por método (porcentagens).

Benchmark	**Média (%)**	**Desvio Padrão**
401.bzip2	49,65	0,00274
429.mcf	50,05	0,00216
433.milc	50,03	0,00073
445.gobmk	50,07	0,00236
458.sjeng	50,01	0,00176
462.libquantum	50,12	0,00214
464.h264ref	50,22	0,00208
470.lbm	49,83	0,00608
482.sphinx3	49,94	0,00112
998.specrand	49,95	0,01514
999.specrand	49,52	0,00764
473.astar	49,91	0,00375

Tabela 7.16. Médias de execução *postpass* por *benchmark* (porcentagens).

As Tabelas 7.15 e 7.16, considerando a soma dos tempos de execução das compilações feitas com o escalonamento *prepass* e das compilações feitas com o escalonamento

postpass, indicam, respectivamente, para cada um dos métodos de compilação e para cada um dos *benchmarks*, a porcentagem média do tempo de execução dos programas compilados com o escalonamento *postpass* em relação à soma feita.

Como indicado nessas tabelas e apresentado nos gráficos da Figura 7.14, o momento de execução da tarefa de escalonamento não interferiu de modo significativo no desempenho dos *benchmarks*. De fato, as compilações com escalonamento *prepass* e as compilações com escalonamento *postpass* tiveram tempos de execução muito próximos e iguais a 50% do tempo total. Além disso, a variância dos tempos medidos foi muito pequena.

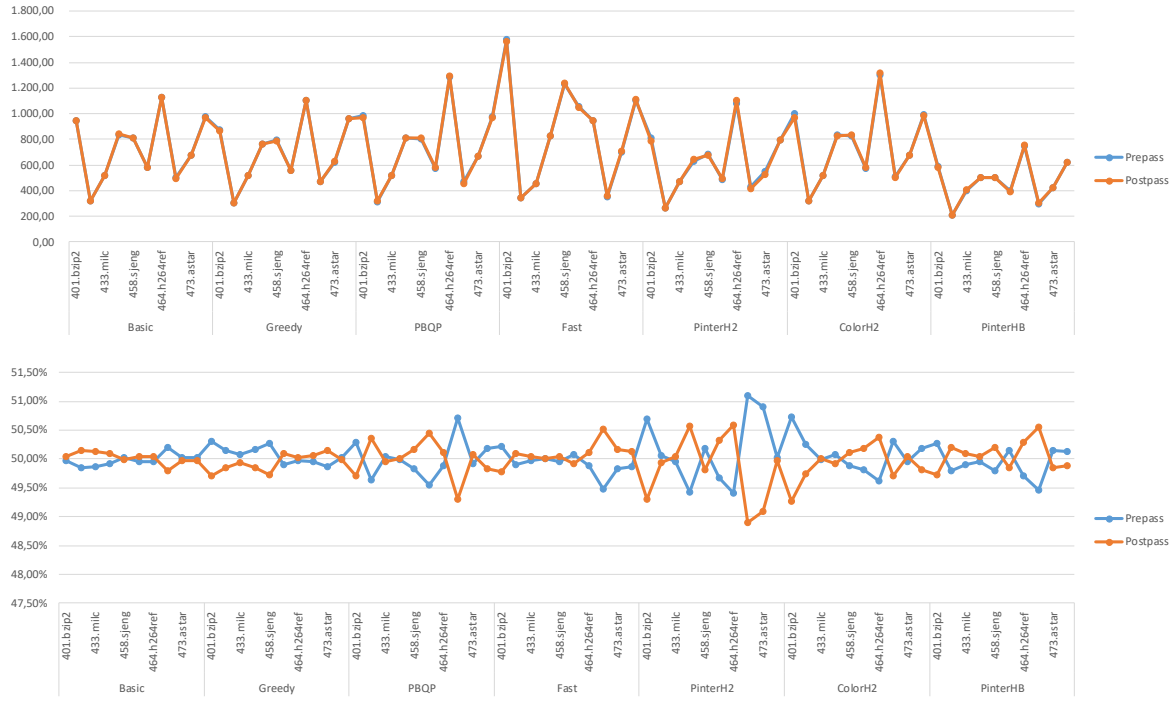


Figura 7.14. Comparação entre os escalonamentos *prepass* e *postpass*.

Este teste comprova o fato de que as arquiteturas que escalonam as instruções dinamicamente e iniciam as instruções fora de ordem, como é o caso da arquitetura utilizada nesses testes, na prática desconsideram a sequência de instruções definida pela tarefa de escalonamento do compilador e fazem com que os desempenhos dos códigos compilados sejam independentes dos resultados e do momento de execução dessa tarefa.

7.3 Análise dos Resultados

Pelos testes realizados, podemos afirmar que a implementação da abordagem de Pinter utilizando a heurística HB para a coloração do grafo de interferência paralelizável foi

capaz de gerar códigos muito eficientes. Com efeito, do ponto de vista do desempenho dos programas, o método *PinterHB* mostrou-se superior aos métodos *PinterH2* e *ColorH2*. Mostrou-se superior aos quatro métodos de alocação de registradores disponíveis no LLVM e, finalmente, também foi superior ao próprio Microsoft Visual Studio, gerando códigos mais rápidos para a plataforma Windows sobre a arquitetura X86.

Uma vez que o microprocessador utilizado nos testes é capaz de iniciar instruções fora de ordem, os escalonamentos efetivamente executados podem ter sido diferentes dos escalonamentos definidos pelo compilador, pois estes últimos podem ter sido alterados pela máquina. Como não é possível conhecer o escalonamento final executado nem mensurar o trabalho necessário à sua produção, não há como afirmar de forma absolutamente precisa que o tratamento dado ao problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções seja o único responsável pelo melhor desempenho dos códigos compilados com a implementação feita. Há de se considerar a possibilidade de que o escalonamento dinâmico tenha contribuído para o melhor resultado observado em algumas situações. A rigor, somente a partir de testes adicionais, realizados com arquiteturas que respeitam o escalonamento definido pelo compilador, tais como as arquiteturas da maioria dos microprocessadores ARM, seria possível tomar a implementação feita como causa única para os melhores desempenhos observados.

Não obstante, os testes realizados fornecem indícios de que a implementação feita é uma boa e adequada solução para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, pois apontam para um estreito relacionamento entre essa implementação e os bons resultados de desempenho obtidos.

O primeiro desses indícios está associado aos próprios resultados de desempenho. Seria de se esperar que o tratamento inadequado do problema gerasse resultados ruins, ou ao menos não tão bons quanto os que foram observados. Certamente, tal tratamento não deveria produzir os melhores resultados. Isso porque ele implicaria, ou uma alocação de registradores ruim, responsável por muitos derramamentos para a memória, ou um escalonamento de instruções ruim, decorrente da criação de muitas falsas dependências entre as instruções. Em ambos os casos, ainda que o escalonamento dinâmico fosse rápido o bastante para compensar a perda de desempenho associada aos derramamentos e bom o bastante para eliminar as falsas dependências criadas entre as instruções, seria de se esperar que o tamanho total do escalonamento aumentasse em determinadas situações, impactando o desempenho dos programas relacionados. Isso, todavia, não foi observado. Dentre os 70 códigos produzidos com o escalonamento *postpass*, todos aqueles que apresentaram melhor desempenho foram compilados com

o método *PinterHB*.

Um segundo indício está relacionado aos derramamentos de valores para a memória. Em última análise, minimizar o número de derramamentos pode ser considerado um dos principais objetivos de um alocador de registradores. Nesse sentido, os resultados do Teste T5 mostram que a abordagem implementada, utilizada em conjunto com a Heurística HB, produziu um bom alocador de registradores. Entretanto, esses mesmos resultados também mostram que o alocador produzido não foi capaz de minimizar o número de derramamentos para todos os programas testados, de tal modo que os bons resultados de desempenho fornecidos pela abordagem *PinterHB* não podem ser atribuídos somente ao baixo número de derramamentos para a memória, ou seja, não decorrem exclusivamente do método de alocação de registradores utilizado. Assim, parece ser razoável supor que essa abordagem também facilitou o processo de escalonamento dinâmico, por meio de uma escolha adequada dos registradores disponíveis na arquitetura. Uma escolha tal que facilitou, ou ao menos não prejudicou o trabalho do escalonador, possivelmente reduzindo a utilização de registradores de nomeação e o número de falsas dependências, que, afinal, também precisam ser resolvidas pelo microprocessador.

```c
#include <stdio.h>

void f(int*, int*);
int main();

int main() {
  int a = 100;
  int b = 1;
  int c = 0;
  f(&a,&b);
  c = a + b;
  printf("a=%d, b=%d, c=%d",
     a, b, c);
  return 0;
}

void f(int* a, int* b) {
  *a = *a - *b;
  while (*a > 1)
    f(a, b);
}
```

(a)

```
_main:
# BB#0:
   subl   $24, %esp
   leal   16(%esp), %eax
   movl   %eax, 4(%esp)
   leal   20(%esp), %eax
   movl   $100, 20(%esp)
   movl   $1, 16(%esp)
   movl   %eax, (%esp)
   calll  _f
   movl   20(%esp), %eax
   leal   1(%eax), %ecx
   movl   %ecx, 12(%esp)
   movl   %eax, 4(%esp)
   movl   $1, 8(%esp)
   movl   $"??_C@...", (%esp)
   calll  _printf
   xorl   %eax, %eax
   addl   $24, %esp
   retl
```

(b)

```
_main:
# BB#0:
   pushl %esi
   subl  $24, %esp
   leal  16(%esp), %esi
   leal  20(%esp), %edx
   movl  $100, 20(%esp)
   movl  $1, 16(%esp)
   movl  %esi, 4(%esp)
   movl  %edx, (%esp)
   calll _f
   movl  20(%esp), %ecx
   leal  1(%ecx), %eax
   movl  %eax, 12(%esp)
   movl  %ecx, 4(%esp)
   movl  $1, 8(%esp)
   movl  $"??_C@...", (%esp)
   calll _printf
   xorl  %eax, %eax
   addl  $24, %esp
   popl  %esi
   retl
```

(c)

Figura 7.15. Códigos gerados: (a) código fonte; (b) código objeto gerado para a função *main* com o método *greedy*; (c) código objeto gerado para a função *main* com o método *PinterHB*.

Tomando como verdadeira a suposição feita, uma das características da escolha

citada seria necessariamente a utilização de um número maior de registradores. Essa característica está de acordo com o processo de construção do grafo de interferência paralelizável e pôde ser observada na grande maioria dos programas analisados. Como exemplo disso, a Figura 7.15 (a) apresenta um código fonte simples, cujas compilações com os métodos *greedy* e *PinterHB* geraram, respectivamente, para a função *main*, os códigos objeto apresentados em (b) e (c). É fácil verificar que, enquanto o código compilado com o método *greedy* utiliza três registradores, *%esp*, *%eax* e *%ecx*, o código produzido com o método *PinterHB* utiliza cinco registradores: os três citados e também os registradores *%esi* e *%edx*.

Indubitavelmente, os testes realizados indicam que a implementação da abordagem de Pinter descrita neste trabalho é uma boa solução para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções considerando a arquitetura CISC utilizada. Os indícios destacados apontam para a possibilidade de que essa implementação seja uma solução de qualidade para esse problema considerando quaisquer arquiteturas. Todavia, tais indícios, ainda que plausíveis e significativos, não são constatações factuais. Para que tais constatações sejam obtidas, como já foi dito anteriormente, seriam convenientes testes adicionais com microprocessadores construídos conforme a arquitetura RISC, ou ao menos com microprocessadores que executam as instruções na ordem definida pelo compilador. Melhor seria, caso um resultado mais preciso fosse desejado para uma arquitetura específica, testar códigos compilados para essa arquitetura.

Finalmente, é importante analisar a implementação feita a partir dos tempos de compilação observados. Realmente, tais tempos inviabilizam a utilização do método desenvolvido em compilações constantes de uma aplicação. Todavia, devido aos resultados obtidos em termos do desempenho do código gerado, o desenvolvimento realizado poderia ser utilizado, justificadamente, como método de compilação otimizada de quaisquer sistemas, ativado por meio de uma opção de compilação específica. Aplicações embarcadas em hardware, por exemplo, tais como aquelas utilizadas em dispositivos móveis, poderiam se beneficiar do método implementado, mesmo com o grande tempo necessário à geração do código executável, uma vez que o código final dessas aplicações poderia ser gerado uma única vez ao término da versão final desenvolvida.

7.4 Considerações Finais

Este capítulo descreveu os testes realizados com a implementação da abordagem de Pinter que foi desenvolvida no LLVM e descrita no Capítulo 6. Os resultados obtidos

nesses testes também foram apresentados de forma detalhada e mostraram que a implementação feita foi capaz de gerar códigos de alto desempenho para as linguagens C e C++.

Os testes utilizaram os *benchmarks* do conjunto SPEC CPU2006, e a abordagem elaborada foi testada com duas heurísticas diferentes para a coloração do grafo de interferência paralelizável. Tais abordagens foram denominadas H2 e HB. Os resultados com a heurística HB foram superiores aos resultados de quaisquer outros métodos testados, incluindo os que foram produzidos pelo próprio Visual Studio da Microsoft. Tal fato pode ser considerado relevante, uma vez que os testes foram realizados sob a plataforma Windows em uma arquitetura X86.

Os resultados obtidos não puderam ser associados de modo inequívoco à solução do problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Isso porque o microprocessador utilizado nos testes realiza o escalonamento dinâmico das instruções e é capaz de iniciá-las fora da ordem definida pelo compilador. Apesar disso, indícios consistentes, apontando a capacidade da implementação feita tratar de modo eficiente o problema da interdependência em quaisquer arquiteturas, puderam ser extraídos dos testes realizados.

Os testes feitos também indicaram que o tempo de compilação com o método implementado é muito alto em todas as situações. Tal fato é um impedimento para a utilização da implementação em compilações constantes e rotineiras. Entretanto, a alta eficiência dos códigos compilados justificaria a utilização dessa implementação para a compilação otimizada de aplicações.

Capítulo 8

Conclusão

Esta dissertação descreveu os trabalhos realizados para a investigação e o tratamento do problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções. Esse problema pode ser visto como um jogo de disputa entre as duas tarefas citadas, de tal modo que a utilização da solução ótima de qualquer uma delas geralmente impede a utilização da solução ótima associada à outra.

O trabalho descrito nesta dissertação contemplou as duas seguintes atividades: uma revisão sistemática da literatura sobre o problema da interdependência; e a implementação, em um compilador real, de um dos métodos identificados na literatura para solucionar esse problema. O compilador utilizado foi o LLVM, e o método implementado foi a abordagem de Pinter, descrita em Pinter [1996].

O problema da interdependência surgiu na década de 1980, juntamente com as arquiteturas RISC e com a transferência para os compiladores de tarefas relevantes para a geração de código. Os resultados deste trabalho mostram que o problema da interdependência, apesar de antigo, tem sido objeto de estudos atuais. Eles comprovam ainda que o tratamento desse problema pode ter consequências bastante positivas no processo de geração de códigos eficientes.

A revisão sistemática da literatura indicou que todas as soluções propostas para o problema da interdependência buscam contemporizar a situação de disputa entre as tarefas de alocação e escalonamento, realizando-as de tal modo que uma leve em consideração as necessidades, características e limitações da outra. Nesse sentido, todas as propostas de solução identificadas realizam essas duas tarefas de modo integrado, ou cooperativo.

A implementação desenvolvida foi testada com os *benchmarks* do conjunto SPEC CPU2006 sob o sistema operacional Microsoft Windows® em uma arquitetura X86. Os códigos compilados com o método implementado tiveram desempenho superior aos

códigos gerados com o LLVM e com o Visual Studio. Em média, aproximadamente, os ganhos em relação ao LLVM foram de 32%, e os ganhos em relação ao Visual Studio foram de 14%. Os resultados dos testes mostraram que a abordagem implementada gerou códigos mais eficientes mesmo para arquiteturas CISC utilizando tecnologias superescalares. Todavia, tais arquiteturas não são as mais afetadas pelo problema e, assim, espera-se que a implementação feita também proporcione resultados positivos em arquiteturas RISC, ou arquiteturas que não utilizem técnicas de iniciação fora de ordem de instruções. Em função da indisponibilidade do *hardware* necessário, testes nessas arquiteturas não foram realizados.

8.1 Contribuições

O trabalho realizado produziu como resultado as seguintes contribuições:

1. Estudo em larga escala sobre o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, capaz de auxiliar futuros pesquisadores a:

   a) melhor compreender o problema;
   b) melhor avaliar os efeitos que ele possui sobre a otimização de código;
   c) melhor conhecer as abordagens utilizadas para lidar com o problema;
   d) melhor entender a relação do problema com as diferentes arquiteturas de computador.

2. Uma fonte alternativa de informações sobre o LLVM, que possa apoiar o processo de entendimento das ferramentas de geração de código desse compilador.
3. Um exemplo real de implementação de um passo de alocação de registradores no LLVM, capaz de ser auxiliar e facilitar futuras implementações.
4. Uma biblioteca escrita em C++ para a manipulação de grafos simples, multigrafos e pseudografos, sejam eles dirigidos ou não dirigidos.
5. Uma fonte de informação sobre a abordagem descrita em Pinter [1996] para tratar o problema da interdependência.
6. Uma implementação real da abordagem descrita em Pinter [1996], capaz de auxiliar futuros pesquisadores a melhor entender aspectos práticos do problema da interdependência, bem como facilitar o entendimento necessário à implementação de outros métodos para o tratamento desse problema.

7. Um método de compilação real capaz de gerar códigos de alto desempenho para arquiteturas CISC com processadores que iniciam instruções fora de ordem.
8. Uma fonte auxiliar de informações sobre os conjuntos de *benchmarks* SPEC CPU, capaz de facilitar o entendimento desses conjuntos e agilizar a sua utilização.
9. Artigo técnico apresentado no VIII CBSoft, XXI SBLP [Carvalho et al., 2017].

8.2 Trabalhos Futuros

Considerando as duas principais atividades que constituem o trabalho descrito nesta dissertação de mestrado, sugerimos os seguintes trabalhos futuros:

Relacionados à revisão sistemática da literatura

1. Construir e testar outras *strings* de busca, formadas por novas palavras-chave.
2. Utilizar outras bases de dados, tais como o Google Acadêmico, no processo de busca dos estudos primários.
3. Verificar as listas de referência dos estudos primários selecionados, a fim de identificar a existência de outros estudos relevantes.
4. Incluir novos critérios de análise dos dados extraídos dos estudos primários, a fim de refinar a comparação entre as diferentes abordagens apresentadas. Como exemplo desses critérios, podemos citar: as áreas e aplicações consideradas nos estudos; a modelagem utilizada na solução; o tempo de execução dos códigos desenvolvidos; o número de *softwares* e trabalhos acadêmicos influenciados.
5. Identificar as circunstâncias que fazem uma abordagem ser melhor do que outra.
6. Quantificar o impacto de cada estudo primário sobre os trabalhos subsequentes, a fim de identificar os estudos mais relevantes e refinar o processo de seleção.

Relacionados à implementação da abordagem de Pinter no LLVM

1. Realizar testes em arquiteturas RISC, ou em arquiteturas que não utilizem a iniciação fora de ordem de instruções. Quaisquer modelos de quaisquer famílias de processadores ARM, tais como ARM7, ARM9, ARM11, Cortex-M, Cortex-R e Cortex-A, com exceção do modelo Cortex-A15 MPCore, podem ser utilizados nesses testes.

2. Considerar as análises de dominância e pós-dominância para identificar os blocos básicos que não podem ser escalonados simultaneamente e refinar o processo de geração do grafo de interferência paralelizável.
3. Aprimorar o tratamento das irregularidades das arquiteturas no código desenvolvido, isto é, o fato dos registradores virtuais e dos registradores da máquina estarem segmentados em classes, de tal modo que um registrador virtual de uma determinada classe tenha de ser atribuído exclusivamente a registrador dessa mesma classe. Por exemplo, um registrador virtual associado a um tipo de ponto flutuante não pode ser atribuído a um registrador de uma classe que lida com tipos inteiros.
4. Tornar as heurísticas desenvolvidas mais robustas, aprimorando o tratamento dado ao código rematerializado após o derramamento de valores para a memória.

   De modo geral, quando um determinado pseudoregistrador é derramado para a memória, o código do programa precisa ser alterado, para que as referências para a memória passem a ser consideradas. No LLVM, isso é feito como ilustrado no exemplo real apresentado na Figura 8.1.

```
16B             %vreg0<def> = MOV32rm <fi#-1>, 1, %noreg, 0, %noreg; mem:LD4[FixedStack-1] GR32:%vreg0
128B            MOV32mr <fi#3>, 1, %noreg, 0, %noreg, %vreg0; mem:ST4[%_Stream.addr] GR32:%vreg0
368B            MOV32mr %ESP, 1, %noreg, 8, %noreg, %vreg0; mem:ST4[Stack+8] GR32:%vreg0

Inline spilling GR32:%vreg0 [16r,368r:0)  @@16r
From original %vreg0
        remat:  120r    %vreg12<def> = MOV32rm <fi#-1>, 1, %noreg, 0, %noreg; mem:LD4[FixedStack-1] GR32:%vreg12
                128e    MOV32mr <fi#3>, 1, %noreg, 0, %noreg, %vreg12<kill>; mem:ST4[%_Stream.addr] GR32:%vreg12

        remat:  360r    %vreg13<def> = MOV32rm <fi#-1>, 1, %noreg, 0, %noreg; mem:LD4[FixedStack-1] GR32:%vreg13
                368e    MOV32mr %ESP, 1, %noreg, 8, %noreg, %vreg13<kill>; mem:ST4[Stack+8] GR32:%vreg13

All defs dead: %vreg0<def,dead> = MOV32rm <fi#-1>, 1, %noreg, 0, %noreg; mem:LD4[FixedStack-1] GR32:%vreg0
Remat created 1 dead defs.
Deleting dead def 16r   %vreg0<def,dead> = MOV32rm <fi#-1>, 1, %noreg, 0, %noreg; mem:LD4[FixedStack-1] GR32:%vreg0
0 registers to spill after remat.
```

Figura 8.1. Rematerialização após *spill code* no LLVM.

Nesse exemplo, as três primeiras linhas pertencem ao *bitcode* analisado. As demais linhas são mensagens enviadas pelo compilador, informando que o registrador virtual *%vreg0* será derramado para a memória e relatando o processo de rematerialização decorrente. Nesse processo, as instruções que utilizam o pseudoregistrador *%vreg0*, cujos índices são 128B e 368B, são substituídas por novas instruções, que utilizam os registradores virtuais *%vreg12* e *%vreg13*. Esses novos pseudoregistradores pertencem à mesma classe de *%vreg0*, no caso, a classe GR32, e são marcados como *kill*, para que o alocador saiba que eles não podem ser derramados para a memória nas futuras etapas de alocação, tendo em vista que,

após a rematerialização, o código alterado é novamente submetido ao alocador, para que todo o processo de alocação de registradores seja refeito.

Desse modo, no grafo de interferência, os vértices associados a esses novos pseudo-registradores são vértices livres. Todavia, durante a construção do grafo de falsas dependências para esse novo código, conforme a abordagem descrita em Pinter [1996], arestas de falsas dependências podem incidir sobre os vértices associados a esses registradores virtuais, fazendo com que eles deixem de ser livres no grafo de interferência paralelizável e assim, teoricamente, possam ser selecionados para um novo derramamento pelo alocador. Tal possibilidade teórica não ocorreu nos testes realizados, em função de questões relacionadas ao custo de derramamento desses pseudoregistradores. No entanto, um tratamento adequado dessa situação talvez possa colaborar para a simplificação do processo de geração do grafo de interferência paralelizável.

5. Avaliar diferentes métodos de escalonamento de instruções, incluindo métodos para o escalonamento global do código, em conjunto com a abordagem de Pinter.

Referências Bibliográficas

Agerwala, T. & Cocke, J. (1987). High performance reduced instruction set processors.

Aho, A. V.; Lam, M. S.; Sethi, R. & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0321486811.

Allen, F. E. (1970). Control flow analysis. Em *Proceedings of a Symposium on Compiler Optimization*, pp. 1–19, New York, NY, USA. ACM.

Almeida, N. M. d. (1979). *Gramática metódica da língua portuguesa*. Edição Saraiva.

Ambrosio, L. L.; Bigonha, M. A. d. S. & Bigonha, R. d. S. (2004). Alocacao global de registradores baseada em crescimento de domínios ativos e combinação de registradores. *VIII Simpósio Brasileiro de Linguagens de Programação*.

Appel, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA. ISBN 0-521-58388-8.

Barany, G. & Krall, A. (2013). Optimal and heuristic global code motion for minimal spilling. Em *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pp. 21–40, Berlin, Heidelberg. Springer-Verlag.

Bergner, P.; Dahl, P.; Engebretsen, D. & O'Keefe, M. (1997). Spill code minimization via interference region spilling. Em *in SIGPLAN Conference on Programming Language Design and Implementation*, pp. 287–295.

Bernstein, D.; Golumbic, M.; Mansour, y.; Pinter, R.; Goldin, D.; Krawczyk, H. & Nahshon, I. (1989). Spill code minimization techniques for optimizing compliers. *SIGPLAN Not.*, 24(7):258–263. ISSN 0362-1340.

Berson, D.; Gupta, R. & Soffa, M. (1994). Resource spackling - a framework for integrating register allocation in local and global schedulers. Em Cosnard, M.; Gao,

G. & Silberman, G., editores, *Parallel Architectures and Compilation Techniques*, volume 50 of *IFIP Transactions A-Computer Science and Technology*, pp. 135–145.

Bradlee, D. G.; Eggers, S. J. & Henry, R. R. (1991). Integrating register allocation and instruction scheduling for riscs. *SIGARCH Comput. Archit. News*, 19(2):122–131. ISSN 0163-5964.

Brasier, T. S.; Sweany, P. H. & Beaty, S. J. (1995). Craig: a practical framework for combining instruction scheduling and register assignment. pp. 11–18. cited By 4.

Briggs, P. (1992). *Register Allocation via Graph Coloring*. Tese de doutorado, Houston, TX, USA. UMI Order No. GAX92-34388.

Briggs, P.; Cooper, K. D.; Kennedy, K. & Torczon, L. (1989). Coloring heuristics for register allocation. Em *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pp. 275–284, New York, NY, USA. ACM.

Briggs, P.; Cooper, K. D. & Torczon, L. (1992). Rematerialization. *SIGPLAN Not.*, 27(7):311–321. ISSN 0362-1340.

Briggs, P.; Cooper, K. D. & Torczon, L. (1994). Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455. ISSN 0164-0925.

Brock, D. C. (2006). *Understanding Moore's Law: Four Decades of Innovation. Edited by David C. Brock.* Chemical Heritage Foundation, Philadelphia. ISBN 9780941901413.

Burger, D.; Austin, T. M. & Bennett, S. (1996). Evaluating future microprocessors: the simplescalar tool set. Relatório técnico.

Carvalho, J. F. N.; Sousa, B. L.; Araújo, M. R. & Bigonha, M. A. S. (2017). The register allocation and instruction scheduling challenge. Em *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP 2017, pp. 3:1–3:9, New York, NY, USA. ACM.

Castro, G. d. (2002). Estrangeirismos: guerras em torno da língua. *Educar em Revista*, pp. 301–306.

Chaitin, G. J. (1982). Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101. ISSN 0362-1340.

Chaitin, G. J.; Auslander, M. A.; Chandra, A. K.; Cocke, J.; Hopkins, M. E. & Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6(1):47 – 57. ISSN 0096-0551.

Chang, C.-M.; Chen, C.-M. & King, C.-T. (1997). Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications*, 34(9):1 – 14. ISSN 0898-1221.

Chao, L.-F. & LaPaugh, A. (1993). Rotation scheduling: A loop pipelining algorithm. Em *Proceedings of the 30th International Design Automation Conference*, DAC '93, pp. 566–572, New York, NY, USA. ACM.

Chen, C.; Novick, G. & Shimano, K. (2000). What is risc? Disponível em http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html. Último acesso em 05/12/2016.

Chow, F. & Hennessy, J. (2004). Register allocation by priority-based coloring. *SIGPLAN Not.*, 39(4):91–103. ISSN 0362-1340.

Cocke, J. & Markstein, V. (2000). The evolution of risc technology at ibm. *IBM J. Res. Dev.*, 44(1-2):48–55. ISSN 0018-8646.

Codina, J. M.; Llosa, J. & González, A. (2002). A comparative study of modulo scheduling techniques. Em *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pp. 97–106, New York, NY, USA. ACM.

Colwell, R. P.; Hitchcock, C. Y. & Jensen, E. D. (1983). Peering through the risc/cisc fog: An outline of research. *SIGARCH Comput. Archit. News*, 11(1):44–50. ISSN 0163-5964.

Cook, T. & Campbell, D. (1979). *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin.

Cooper, K. D. & Taylor Simpson, L. (1998). *Live range splitting in a graph coloring register allocator*, pp. 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edição. ISBN 0262033844, 9780262033848.

Corporation, M. (2017a). Microsoft developer network. Disponível em https://msdn.microsoft.com. Último acesso em 10/12/2017.

Corporation, S. P. E. (2017b). Standard performance evaluation corporation. Disponível em https://www.spec.org. Último acesso em 10/12/2017.

Cunha, C. F. d. (1975). *Língua portuguesa e realidade brasileira*. Tempo Brasileiro.

Cutcutache, I. & Wong, W.-F. (2008). Fast, frequency-based, integrated register allocation and instruction scheduling. *Softw. Pract. Exper.*, 38(11):1105–1126. ISSN 0038-0644.

Dandamudi, S. P. (2005). *RISC Principles*, pp. 39–44. Springer New York, New York, NY.

Depuydt, F.; Goossens, G. & Man, H. D. (1994). Scheduling with register constraints for dsp architectures. *Integration, the VLSI Journal*, 18(1):95 – 120. ISSN 0167-9260.

Domagala, L.; van Amstel, D.; Rastello, F. & Sadayappan, P. (2016). Register allocation and promotion through combined instruction scheduling and loop unrolling. Em *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pp. 143–151, New York, NY, USA. ACM.

Fink, A. (2010). *Conducting Research Literature Reviews: From the Internet to Paper*. SAGE Publications. ISBN 9781412971898.

Fisher, J. A. (1981). Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490. ISSN 0018-9340.

Fisher, J. A.; Ellis, J. R.; Ruttenberg, J. C. & Nicolau, A. (1984). Parallel processing: A smart compiler and a dumb machine. *SIGPLAN Not.*, 19(6):37–47. ISSN 0362-1340.

Gao, L. & Shi, C. (2005). An improved approach of register allocation via graph coloring.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA. ISBN 0716710447.

Gibbons, P. B. & Muchnick, S. S. (1986). Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.*, 21(7):11–16. ISSN 0362-1340.

Goodman, J. R. & Hsu, W.-C. (1988). Code scheduling and register allocation in large basic blocks. Em *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pp. 442–452, New York, NY, USA. ACM.

Govindarajan, R.; Yang, H.; Amaral, J. N.; Zhang, C. & Gao, G. R. (2003). Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Transactions on Computers*, 52(1):4–20. ISSN 0018-9340.

Govindarajan, R.; Zhang, C. & Gao, G. R. (2000). Minimum register instruction scheduling: A new approach for dynamic instruction issue processors. Em *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pp. 70–84, London, UK, UK. Springer-Verlag.

Heath, J. L. (1984). Re-evaluation of the risc i. *SIGARCH Comput. Archit. News*, 12(1):3–10. ISSN 0163-5964.

Hendren, L. J.; Gao, G. R.; Altman, E. R. & Mukerji, C. (1992). A register allocation framework based on hierarchical cyclic interval graphs. Em *Proceedings of the 4th International Conference on Compiler Construction*, CC '92, pp. 176–191, London, UK, UK. Springer-Verlag.

Hennessy, J.; Jouppi, N.; Baskett, F.; Gross, T. & Gill, J. (1982). Hardware/software tradeoffs for increased performance. *SIGPLAN Not.*, 17(4):2–11. ISSN 0362-1340.

Hennessy, J. L. (1984). Vlsi processor architecture. *IEEE Transactions on Computers*, C-33(12):1221–1246. ISSN 0018-9340.

Hennessy, J. L. & Gross, T. (1983). Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448. ISSN 0164-0925.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17. ISSN 0163-5964.

Huang, Y.; Li, Q. & Xue, C. J. (2011). Minimizing schedule length via cooperative register allocation and loop scheduling for embedded systems. Em *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '11, pp. 1038–1044, Washington, DC, USA. IEEE Computer Society.

Huff, R. A. (1993). Lifetime-sensitive modulo scheduling. Em *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pp. 258–267, New York, NY, USA. ACM.

Infocenter, A. (2017). Arm (advanced risc machine). Disponível em http://infocenter.arm.com. Último acesso em 10/12/2017.

Ivanov, D. S. (2010). Register allocation with instruction scheduling for vliw-architectures. *Program. Comput. Softw.*, 36(6):363–367. ISSN 0361-7688.

Johnson, M. (1991). *Superscalar microprocessor design*. Prentice Hall series in innovative technology. Prentice Hall. ISBN 9780138756345.

Josey, A. (2017). Posix - austin joint working group. Disponível em http://standards.ieee.org/develop/wg/POSIX.html. Último acesso em 10/12/2017.

Jung, C. (2004). *Metodologia para pesquisa e desenvolvimento: aplicada a novas tecnologias, produtos e processos*. Axcel Books. ISBN 9788573232332.

Karp, R. M. (1972). *Reducibility among Combinatorial Problems*, pp. 85–103. Springer US, Boston, MA.

Katevenis, M. G. H. (1985). *Reduced Instruction Set Computer Architectures for VLSI*. Massachusetts Institute of Technology, Cambridge, MA, USA. ISBN 0-262-11103-9.

Kessler, C. W. (1998). Scheduling expression dags for minimal register need. *Computer Languages*, 24(1):33 – 53. ISSN 0096-0551.

Kim, D.-H. & Lee, H.-J. (2010). Fine-grain register allocation and instruction scheduling in a reference flow. *Comput. J.*, 53(6):717–740. ISSN 0010-4620.

Kiran, D.; Gurunarayanan, S.; Misra, J. P. & Bhatia, M. (2015). Register allocation for fine grain threads on multicore processor. *Journal of King Saud University - Computer and Information Sciences*, pp. –. ISSN 1319-1578.

Kiran, D. C.; Gurunarayanan, S.; Khaliq, F. & Nawal, A. (2012). *Compiler Efficient and Power Aware Instruction Level Parallelism for Multicore Architecture*, pp. 9–17. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kiran, D. C.; Gurunarayanan, S. & Misra, J. P. (2011a). Taming compiler to work with multicore processors. Em *2011 International Conference on Process Automation, Control and Computing*, pp. 1–6.

Kiran, D. C.; Radheshyam, B.; Gurunarayanan, S. & Misra, J. P. (2011b). Compiler assisted dynamic scheduling for multicore processors. Em *2011 International Conference on Process Automation, Control and Computing*, pp. 1–6.

Kitchenham, B. (2004). Procedures for performing systematic reviews. Relatório técnico, Keele University and NICTA.

Kitchenham, B. & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Relatório técnico EBSE 2007-001, Keele University and Durham University Joint Report.

Kolte, P. & Harrold, M. J. (1993). Load/store range analysis for global register allocation. *SIGPLAN Not.*, 28(6):268–277. ISSN 0362-1340.

Koseki, A.; Komatsu, H. & Nakatani, T. (2003). Spill code minimization by spill code motion. Em *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 125–134. ISSN 1089-795X.

Kri, F. & Feeley, M. (2004). Genetic instruction scheduling and register allocation. Em *Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, QEST '04, pp. 76–83, Washington, DC, USA. IEEE Computer Society.

Lattner, C. & Adve, V. (2003). Llvm: A compilation framework for lifelong program analysis & transformation. Tech. Report UIUCDCS-R-2003-2380, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

Lattner, C. & Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. Em *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pp. 75–, Washington, DC, USA. IEEE Computer Society.

Leiserson, C. E.; Rose, F. M. & Saxe, J. B. (1983). *Optimizing Synchronous Circuitry by Retiming (Preliminary Version)*, pp. 87–116. Springer Berlin Heidelberg, Berlin, Heidelberg.

Leupers, R. (2000). *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Kluwer Academic Publishers, Norwell, MA, USA. ISBN 0792379896.

LINDO Systems, I. (2003). *LINDO (Linear, INteractive, and Discrete Optimizer) - User's Manual*. Chicago, Illinois, USA.

Lo, R.; Chow, F.; Kennedy, R.; Liu, S.-M. & Tu, P. (1998). Register promotion by sparse partial redundancy elimination of loads and stores. *SIGPLAN Not.*, 33(5):26–37. ISSN 0362-1340.

Lopes, B. C. & Auler, R. (2014). *Getting Started with LLVM Core Libraries*. Packt Publishing. ISBN 1782166920, 9781782166924.

Lozano, R. C. n.; Carlsson, M.; Blindell, G. H. & Schulte, C. (2016). Register allocation and instruction scheduling in unison. Em *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pp. 263–264, New York, NY, USA. ACM.

Lozano, R. C. n.; Carlsson, M.; Drejhammar, F. & Schulte, C. (2012). Constraint-based register allocation and instruction scheduling. Em *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming - Volume 7514*, pp. 750–766, New York, NY, USA. Springer-Verlag New York, Inc.

Luna, M. F.; Silva, F. L. & Attrot, W. (2015). Decreasing spill code to decrease energy consumption. Em *2015 Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 128–131.

M. Bigonha, A. d. S. (1992). Geração e otimização de código: Levantamento dos problemas e restrições impostas pelas arquiteturas risc e indicativos de soluções. *Série de Monografias em Ciência da Computação 10/92*.

M. Bigonha, A. d. S. (1994). *Otimização de código em máquinas superescalares*. Tese de doutorado, Rio de Janeiro, RJ, Brasil.

Mian, P.; Conte, T.; Natali, A.; Biolchini, J. & Travassos, G. (2005). A systematic review process for software engineering. Em *ESELAW '05: 2nd Experimental Software Engineering Latin American Workshop*.

Mizrahi, V. V. (1994). *Treinamento em Linguagem C++: Módulo 2*. Makron Books do Brasil. ISBN 9788534603034.

Mizrahi, V. V. (2006). *Treinamento em Linguagem C++: Módulo 1*. Pearson. ISBN 9788576050452.

Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85. ISSN 0018-9219.

Motwani, R.; Palem, K. V.; Sarkar, V. & Reyen, S. (1995). Combining register allocation and instruction scheduling. Relatório técnico, Stanford, CA, USA.

Ning, Q. & Gao, G. R. (1993). A novel framework of register allocation for software pipelining. Em *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pp. 29–42, New York, NY, USA. ACM.

Norris, C. & Pollock, L. L. (1995). Register allocation sensitive region scheduling. pp. 1–10.

Norris, C. & Pollock, L. L. (1998). Experiences with cooperating register allocation and instruction scheduling. *Int. J. Parallel Program.*, 26(3):241–283. ISSN 0885-7458.

Padua, D. A. & Wolfe, M. J. (1986). Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201. ISSN 0001-0782.

Pandey, M. & Sarda, S. (2015). *LLVM Cookbook*. Packt Publishing. ISBN 178528598X, 9781785285981.

Patterson, D. A. & Piepho, R. S. (1982). Risc assessment: A high-level language experiment. *SIGARCH Comput. Archit. News*, 10(3):3–8. ISSN 0163-5964.

Patterson, D. A. & Sequin, C. H. (1982). A vlsi risc. *Computer*, 15(9):8–21. ISSN 0018-9162.

Petersen, K.; Feldt, R.; Mujtaba, S. & Mattsson, M. (2008). Systematic mapping studies in software engineering. Em *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pp. 68–77, Swindon, UK. BCS Learning & Development Ltd.

Philippidis, C. J. & Shang, W. (2010). On minimizing register usage of linearly scheduled algorithms with uniform dependencies. *Comput. Lang. Syst. Struct.*, 36(3):250–267. ISSN 1477-8424.

Pinter, S. S. (1993). Register allocation with instruction scheduling. Em *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pp. 248–257, New York, NY, USA. ACM.

Pinter, S. S. (1996). Register allocation with instruction scheduling: a new approach. *JOURNAL OF PROGRAMMING LANGUAGES*, 4(1):21–38. ISSN 0963-9306.

Quintão Pereira, F. M. & Palsberg, J. (2008). Register allocation by puzzle solving. *SIGPLAN Not.*, 43(6):216–226. ISSN 0362-1340.

Radin, G. (1982). The 801 minicomputer. *SIGPLAN Not.*, 17(4):39–47. ISSN 0362-1340.

Rau, B. R. (1994). Iterative modulo scheduling: An algorithm for software pipelining loops. Em *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pp. 63–74, New York, NY, USA. ACM.

Rau, B. R.; Lee, M.; Tirumalai, P. P. & Schlansker, M. S. (1992). Register allocation for modulo scheduled loops: Strategies, algorithms and heuristics. *SIGPLAN Not.*, 27(7):283–299.

Ricarte, I. L. M. (1999). Organização de computadores. Disponível em ftp://ftp.dca.fee.unicamp.br/pub/docs/ea960/ea960.pdf. Último acesso em 10/12/2017.

Rimey, K. E. (1989). *A compiler for application-specific signal processors*. Tese de doutorado, Berkeley, CA, USA.

Sarda, S. & Pandey, M. (2015). *LLVM Essentials*. Packt Publishing. ISBN 1785280805, 9781785280801.

Silva, F. L. (2015). Color flipping : minimização de spill code via troca de cores em um grafo de interferência. Dissertação de mestrado.

Stallings, W. (2010). *Computer Organization and Architecture: Designing for Performance*. Prentice Hall. ISBN 9780136073734.

Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional, 4th edição. ISBN 0321563840, 9780321563842.

Tanenbaum, A. S. (2005). *Structured Computer Organization (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0131485210.

Team, L. A. (2017). The llvm compiler infrastructure. Disponível em http://llvm.org. Último acesso em 13/12/2017.

Torczon, L. & Cooper, K. (2011). *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edição. ISBN 012088478X.

Valadares, F. B. (2014). Variação e mudança linguística: uma análise da ampliação semântica de estrangeirismos no português brasileiro. 11:403.

Valluri, M. G. & Govindarajan, R. (1999). Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. Em *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pp. 78–, Washington, DC, USA. IEEE Computer Society.

van Beek, P. & Wilken, K. (2001). *Fast Optimal Instruction Scheduling for Single-Issue Processors with Arbitrary Latencies*, pp. 625–639. Springer Berlin Heidelberg, Berlin, Heidelberg.

Wang, K.; Zhihong, T.; Edwin, Y. & Sha, H. M. (1998). Rcrs: a framework for loop scheduling with limited number of registers. Em *Proceedings of the 8th Great Lakes Symposium on VLSI (Cat. No.98TB100222)*, pp. 386–391. ISSN 1066-1395.

Wohlin, C.; Runeson, P.; Hst, M.; Ohlsson, M. C.; Regnell, B. & Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated. ISBN 3642290434, 9783642290435.

Zalamea, J.; Llosa, J.; Ayguadé, E. & Valero, M. (2001). Modulo scheduling with integrated register spilling for clustered vliw architectures. Em *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pp. 160–169, Washington, DC, USA. IEEE Computer Society.

Zhang, L.; Qiu, M.; Sha, E. H.-M. & Zhuge, Q. (2011). Variable assignment and instruction scheduling for processor with multi-module memory. *Microprocessors and Microsystems*, 35(3):308 – 317. ISSN 0141-9331.