Exploração de sequências de otimização do
compilador baseada em técnicas híbridas de
mineração de dados complexos

Luiz Gustavo Almeida Martins

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Luiz Gustavo Almeida Martins

Exploração de sequências de otimização do compilador baseada em técnicas híbridas de mineração de dados complexos

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Eduardo Marques

Coorientador: Prof. Dr. Alexandre C. B. Delbem

USP – São Carlos
Novembro de 2015

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

M379e Martins, Luiz Gustavo Almeida
Exploração de sequências de otimização do
compilador baseada em técnicas híbridas de mineração
de dados complexos / Luiz Gustavo Almeida Martins;
orientador Eduardo Marques; co-orientador Alexandre
Cláudio Botazzo Delbem. -- São Carlos, 2015.
152 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2015.

1. Compiladores. 2. Agrupamento. 3. Exploração do
espaço de projeto. 4. Problema de ordenação dos
passos de otimização. 5. Sistemas embarcados. I.
Marques, Eduardo, orient. II. Botazzo Delbem,
Alexandre Cláudio, co orient. III. Título.

Luiz Gustavo Almeida Martins

**Exploration of optimization sequences of the compiler
based on hybrid techniques of complex data mining**

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques
Coadvisor: Prof. Dr. Alexandre C. B. Delbem

**USP – São Carlos
November 2015**

Agradecimentos

Agradeço primeiramente a Deus, pelas dádivas que me foram concedidas durante toda a vida.

À minha esposa Gina pelo sacrifício despendido para que eu pudesse me dedicar aos meus estudos e pesquisa, bem como por sempre acreditar em meu potencial, incentivando-me na busca de novos desafios e apoiando-me na superação dos obstáculos.

À minha filha Giovanna, cuja existência enche minha vida de felicidade, por suportar minha ausência e indisponibilidade durante todo esse período, e continuar me dando seu carinho e amor incondicionais.

Aos meus pais e minha vó Ivette pelo carinho, amor e ensinamentos que foram essenciais para a formação do meu ser.

Aos meus irmãos, familiares e amigos, que de forma direta ou indiretamente, sempre estiveram presentes nos principais momentos de minha vida.

Ao meu orientador Prof. Dr. Eduardo Marques, não só pelos ensinamentos e orientação durante a pesquisa, mas também por sua preocupação com meu bem-estar e de minha família, dando sábios conselhos nos momentos mais difíceis.

Ao meu coorientador Prof. Dr. Alexandre C. B. Delbem pelas ideias e bate-papos ao longo do caminho.

Ao Prof. Dr. João M. P. Cardoso por me acolher tão bem em seu laboratório na FEUP e compartilhar toda a infraestrutura e conhecimento necessários à evolução de minha pesquisa.

Aos demais professores do ICMC, pelos ensinamentos e experiências, me proporcionando as condições necessárias à realização deste trabalho.

Aos meus colegas de FACOM, do ICMC e da FEUP, pela cordial convivência e troca de conhecimentos. Em especial, ao Erinaldo, Cristiano, Pedro Pinto e Ricardo Nobre pela ajuda, parceria e discussões ao longo do doutorado.

Aos funcionários da Pós-Graduação do ICMC, pela ajuda oferecida, sempre que necessário.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa concedida pelo Programa Ciência sem Fronteiras (Processo: 0352/13-6), possibilitando meu estágio em Portugal, o qual foi essencial à minha pesquisa.

E, por fim, à Universidade Federal de Uberlândia, em especial à Faculdade de Computação, pelo apoio e liberação.

Resumo

Devido ao grande número de otimizações fornecidas pelos compiladores modernos e à ampla possibilidade de ordenação dessas transformações, uma eficiente Exploração do Espaço de Projeto (DSE) se faz necessária para procurar a melhor sequência de otimização de uma determinada função ou fragmento de código. Como esta exploração é uma tarefa complexa e dispendiosa, apresentamos uma nova abordagem de DSE capaz de reduzir esse tempo de exploração e selecionar sequências de otimização que melhoraram o desempenho dos códigos transformados. Nossa abordagem utiliza um conjunto de funções de referência, para as quais uma representação simbólica do código (DNA) e a melhor sequência de otimização são conhecidas. O DSE de novas funções é baseado em uma abordagem de agrupamento aplicado sobre o código DNA que identifica similaridades entre funções. O agrupamento utiliza três técnicas para a mineração de dados: distância de compressão normalizada, algoritmo de reconstrução de árvores filogenéticas (*Neighbor Joining*) e identificação de grupos por ambiguidade. As otimizações das funções de referência identificadas como similares formam o espaço que é explorado para encontrar a melhor sequência para a nova função. O DSE pode utilizar o conjunto reduzido de otimizações de duas formas: como o espaço de projeto ou como a configuração inicial do algoritmo. Em ambos os casos, a adoção de uma pré-seleção baseada no agrupamento permite o uso de algoritmos de busca simples e rápidos. Os resultados experimentais revelam que a nova abordagem resulta numa redução significativa no tempo total de exploração, ao mesmo tempo que alcança um desempenho próximo ao obtido através de uma busca mais extensa e dispendiosa baseada em algoritmos genéticos.

Palavras-chave: Compiladores, agrupamento, exploração do espaço de projeto, problema de ordenação dos passos de otimização, sistemas embarcados.

Abstract

Due to the large number of optimizations provided in modern compilers and to compiler optimization specific opportunities, a Design Space Exploration (DSE) is necessary to search for the best sequence of compiler optimizations for a given code fragment (e.g., function). As this exploration is a complex and time consuming task, we present new DSE strategies to reduce the exploration time and still select optimization sequences able to improve the performance of each function. The DSE is based on a clustering approach which groups functions with similarities and then explore the reduced search space provided by the optimizations previously suggested for the functions in each group. The identification of similarities between functions uses a data mining method which is applied to a symbolic representation of the source code. The DSE strategies uses the reduced optimizations set identified by clustering in two ways: as the design space or as the initial configuration of the algorithm. In both ways, the adoption of a pre-selection based on clustering allows the use of simple and fast DSE algorithms. Several experiments for evaluating the effectiveness of the proposed approach address the exploration of compiler optimization sequences. Besides, we investigate the impact of each technique or component employed in the selection process. Experimental results reveal that the use of our new clustering-based DSE approach achieved a significant reduction on the total exploration time of the search space at the same time that obtained performance speedups close to a traditional genetic algorithm-based approach.

keywords: Compilers, clustering, design space exploration, phase-ordering problem, embedded systems.

Lista de Figuras

2.1	Estrutura em alto nível de um compilador simples (Muchnick, 1997).	11
2.2	Estrutura típica de um compilador multi-plataforma e multi-linguagens. . . .	13
2.3	Estrutura de um compilador otimizante clássico.	14
2.4	Estrutura de um compilador otimizante adaptativo.	14
2.5	Estrutura de um compilador para <i>hardware</i> reconfigurável (Cardoso e Diniz, 2008).	15
2.6	Visão geral da decidibilidade do problema de ordenação.	22
3.1	Comparativo entre o fluxo manual e automático de DSE.	26
3.2	Exploração em função da estratégia de busca adotada.	31
4.1	Seleção dos passos de otimização do compilador baseado em agrupamento. . . .	46
4.2	Algoritmo de seleção de passos de compilação baseado em agrupamento. . . .	47
4.3	Exemplos de codificação do DNA a partir do código C.	49
4.4	Exemplo da reconstrução de árvore filogenética através do Algoritmo <i>Neighbor Joining</i>	52
4.5	Agrupamento baseado no conceito de ambiguidade forte.	53
4.6	Subárvores formadas pelo agrupamento com até três nós intermediários. . . .	54
5.1	Funcionamento típico de um Algoritmo Genético.	56
5.2	Operador de <i>crossover</i> aplicado pelo AG.	61
5.3	Operadores de mutação aplicados pelo AG.	62
5.4	Algoritmo de <i>Inserção</i>	64
5.5	Algoritmo de <i>Limpeza</i>	65
5.6	Exemplo de funcionamento das abordagens híbridas.	67
5.7	Fluxo de compilação baseado em LARA.	68
6.1	Diagrama de blocos do processador Xilinx Microblaze (Xilinx, 2012).	74
6.2	Árvores filogenéticas geradas a partir de códigos similares aos de algumas funções do conjunto de referência.	79
6.3	Evolução da média geométrica do <i>speedup</i> e do tempo de execução em função do tamanho das sequências usada pelo AG.	84
6.4	Melhor <i>speedup</i> obtido para cada função alvo usando estratégias de DSE baseadas em um único algoritmo de busca sobre os espaços reduzido e completo.	87
6.5	<i>Speedups</i> obtidos para cada função alvo usando estratégias de DSE baseadas em algoritmos híbridos e a sequência de compilação que melhor otimizou todas as funções do conjunto de referência.	90

6.6	Tamanho dos espaços de busca iniciais e das sequências geradas de acordo com a estratégia de DSE e com a função alvo avaliada.	91
6.7	Média geométrica dos melhores <i>speedups</i> de acordo com a estratégia de DSE utilizada.	92
6.8	Tempo de execução das estratégias de DSE no espaço completo ou reduzido obtidos para cada função alvo.	95
6.9	Média geométrica do tempo de execução das estratégias de DSE.	96
6.10	Desempenho das estratégias de DSE avaliadas com um enfoque multi-objetivo.	99
6.11	Média geométrica dos <i>speedups</i> de acordo com o grupo usado pelo DSE.	101
6.12	Melhor <i>speedup</i> alcançado pelo DSE-CI para as funções nas quais o método de seleção não atingiu o menor valor de latência dentre os grupos avaliados.	102
6.13	Árvores filogenéticas geradas a partir das diferentes fontes de entrada para a função <i>autcor</i> —.	106
6.14	Desempenho das estratégias de DSE em função do tipo de representação do código usado como entrada para o método.	107
6.15	<i>Speedup</i> médio obtido pelo DSE baseado no algoritmo de limpeza de acordo com o conjunto de regras de transformação e sua forma de aplicação durante a codificação do DNA.	111
6.16	<i>Speedups</i> médios obtido pelo DSE baseado no algoritmo de limpeza de acordo com a métrica empregada no cálculo das matrizes de distâncias.	116
6.17	Comportamento dos conjuntos avaliados de acordo com o critério utilizado para a seleção das funções de referência.	122
6.18	<i>Speedups</i> obtidos para cada função alvo através das estratégias de DSE não dominadas e usando o passo de compilação <i>loop unrolling</i>	129
6.19	<i>Speedups</i> alcançados para cada novo <i>benchmark</i> usando as estratégias de DSE não-dominadas.	130
6.20	<i>Speedups</i> alcançados no ambiente de compilação do LLVM usando as estratégias DSE-Limp e DSE-CI.	133

Lista de Tabelas

3.1	Características do problema abordado pelos trabalhos correlatos.	37
3.2	Características da abordagem adotada em cada um dos trabalhos correlatos.	39
4.1	Matriz de distâncias NCD a partir da representação DNA.	50
5.1	Passos de compilação do CoSy suportados pelo ambiente de compilação.	69
6.1	Funções do conjunto de referência.	77
6.2	Funções alvo retiradas do repositório de DSP da Texas Instruments.	80
6.3	Funções alvo retiradas do repositório de processamento de vídeo e imagens da Texas Instruments.	81
6.4	Desempenho das estratégias de DSE de acordo com o número de funções.	94
6.5	Agrupamentos utilizados nos experimentos.	101
6.6	Desempenho obtido a partir das abordagens baseadas nas sequências de otimização usadas pelas funções de referência do grupo da função alvo a ser compilada.	103
6.7	Regras de transformação empregadas na codificação do DNA.	108
6.8	Desempenho dos compressores em relação aos <i>speedups</i> médios obtidos pelas abordagens DSE e à sua capacidade de compactação.	113
6.9	Agrupamento de todas as funções manipuladas em 11 grupos distintos pelo algoritmo <i>Fast Newman</i>	120
6.10	Funções com as maiores sequências de cada um dos 11 grupos gerados.	121
6.11	Métricas estatísticas geradas a partir dos conjuntos avaliados em cada estratégia de seleção das funções de referência.	122
6.12	Desempenho das duas abordagens de DSE-AG usando um espaço completo com <i>loop unrolling</i>	125
6.13	Latência obtida a partir da aplicação de sequências com e sem <i>loop unrolling</i> sobre as funções <i>fdct</i> e <i>fdct</i> —.	127
6.14	Média geométrica dos <i>speedups</i> obtidos pelas estratégias de DSE em função da utilização do <i>loop unrolling</i>	128

Sumário

1	Introdução	1
1.1	Motivação	3
1.2	Objetivos	5
1.3	Organização do Texto	6
2	Compilação e Otimização	9
2.1	Compiladores	9
2.1.1	Estrutura básica do compilador	10
2.1.2	Otimizações	12
2.1.3	Compiladores otimizantes	13
2.1.4	Compiladores e sistemas embarcados	15
2.2	Problema da Ordenação dos Passos de Otimização	17
2.2.1	Formalização do problema e aspectos teóricos relacionados	19
2.2.2	Espaço de Busca	23
3	Exploração das Sequências de Otimização	25
3.1	Processo de DSE	26
3.2	Espaço de Projeto	27
3.2.1	Espaço do problema	27
3.2.2	Espaço de objetivos	28
3.3	Redução do Espaço de Busca	28
3.4	Métodos de Exploração	30
3.5	Métodos de Avaliação	32
3.6	Trabalhos Correlatos	34
4	Seleção Baseada em Agrupamento	45
4.1	Codificação do DNA	46
4.2	Distância de Compressão Normalizada	48
4.3	Algoritmo de reconstrução de árvores filogenéticas	50
4.4	Agrupamento Baseado em Ambiguidade	51
5	Estratégias de DSE	55
5.1	Algoritmos Genéticos	55
5.1.1	Seleção de indivíduos	57
5.1.2	<i>Crossover</i>	58
5.1.3	Configuração do AG	59
5.2	Amostragem Aleatória	62
5.3	Algoritmo de Inserção	63
5.4	Algoritmo de Limpeza	65

5.5	Abordagens Híbridas	66
5.6	Infraestrutura do DSE	68
5.6.1	Linguagem LARA	68
5.6.2	Ferramentas de compilação	69
5.6.3	Simulador	71
6	Experimentos	73
6.1	Configuração do Ambiente dos Experimentos	73
6.1.1	Processador MicroBlaze	74
6.1.2	Conjunto de referência	75
6.1.3	Funções alvo (<i>benchmarks</i>)	77
6.2	Metodologia de Avaliação	82
6.3	Configuração dos Parâmetros do Algoritmo Genético	83
6.4	Resultados Experimentais	85
6.4.1	Avaliação das estratégias de DSE	85
6.4.2	Avaliação da eficácia do Método de Agrupamento	100
6.4.3	Análise das representações de entrada	104
6.4.4	Análise das regras de codificação do DNA	108
6.4.5	Análise da matriz de dissimilaridades	112
6.4.6	Análise do conjunto de referência	117
6.4.7	Desdobramento de laços (<i>loop unrolling</i>)	124
6.4.8	Avaliação de novos <i>benchmarks</i>	130
6.4.9	Estendendo a análise comparativa	131
7	Conclusões	135
7.1	Contribuições	138
7.2	Artigos Originados da Tese	140
7.3	Trabalhos Futuros	141
	Referências	143

Capítulo 1

Introdução

A qualidade do código executado influencia diretamente o desempenho final de uma tarefa computacional. Normalmente, esses códigos são escritos em uma linguagem de programação de alto nível, como a linguagem C, e posteriormente convertidos por um compilador em um código que pode ser entendido pela arquitetura alvo, também conhecido como código de máquina. Dessa forma, o compilador exerce um papel importante na qualidade do código executado e, conseqüentemente, no desempenho atingido. Muitos problemas que um compilador resolve durante os diferentes estágios de conversão do código são bastante complexos, em especial, aqueles ligados à etapa de otimização do código (Beg, 2013). Otimizações do compilador tentam minimizar o consumo de recursos pelo código compilado, visando melhorar as métricas de desempenho, tais como, tempo de execução e consumo de potência. Para tal, um conjunto de transformações de código, também conhecidas como passos de otimização ou simplesmente otimizações, deve ser selecionado e sua ordem de aplicação pelo compilador deve ser definida.

Uma prática comum no projeto de compiladores é aplicar um mesmo conjunto de otimizações em uma ordem fixa para cada fragmento de código e/ou função de um programa ao se utilizar uma arquitetura ou plataforma alvo. Entretanto, compiladores para sistemas computacionais embarcados são mais dependentes das otimizações, uma vez que suas arquiteturas têm tipicamente mais restrições relacionadas ao tamanho e organização da memória, velocidade do processador e podem ter altos níveis de personalização, tais como aqueles permitidos no uso de FPGAs (*Field Programmable Gate Array*) (Cardoso et al, 2010). Assim, a escolha do grupo de otimizações, dentre as disponibilizadas pelo compilador, e a ordem pela qual estas serão aplicadas pode ter um impacto significativo no desempenho do código compilado. Além disso, o efeito das otimizações é influenciado pela plataforma utilizada e pela própria aplicação. Sequências de otimização não triviais são usualmente necessárias para atingir os resultados desejados e mesmo os compiladores para FPGAs no estado-da-arte (por exemplo, compiladores de C para HDL - *Hardware Description Language*) necessitam que o usuário selecione manualmente a maioria das sequências de otimização para alcançar resultados eficientes (Buyukkurt et al, 2010). Para

esses sistemas, uma vasta exploração do espaço de projeto (ou, em inglês, *Design Space Exploration* - DSE) deve ser realizada.

Devido à quantidade de transformações implementadas nos compiladores atuais, o tamanho do espaço de busca das possíveis sequências de otimização costuma ser imenso, de modo que sua completa exploração não é uma alternativa viável (Touati e Barthou, 2006). Mesmo desenvolvedores mais experientes demandam muito esforço e tempo para explorar as várias soluções potenciais, tornando a exploração manual um processo muito demorado e trabalhoso. O uso de abordagens heurísticas tradicionais normalmente demanda um alto custo computacional até convergir para uma boa solução (Kulkarni, 2007) (Jantz e Kulkarni, 2013). Portanto, a investigação de técnicas para realizar a busca nesse espaço de forma eficiente e efetiva é um tópico de pesquisa bastante relevante e atual, principalmente na área de sistemas embarcados, onde pode ser considerado um *hot topic* (Kulkarni et al, 2010)(Dragomir, 2011)(Fursin et al, 2011)(Beg, 2013)(Jantz e Kulkarni, 2013)(Martins et al, 2014b)(Martins et al, 2014a).

Nesta tese, nós apresentamos uma nova abordagem de DSE baseada no agrupamento do código para pesquisar sequências de otimização, visando melhorar o desempenho de fragmentos de código. O processo de agrupamento permite a seleção de um conjunto reduzido de passos de otimização, os quais são então usados no processo de DSE para gerar as sequências. Esse agrupamento pode revelar padrões similares entre os códigos de *software*, fornecendo importantes indícios para determinar os potenciais grupos de otimizações.

Nossa abordagem de DSE adota algoritmos simples de busca, que atuam sobre esse conjunto reduzido de otimizações potenciais para encontrar uma boa sequência (ótima ou sub-ótima) capaz de melhorar o desempenho do código compilado. Nossa abordagem de agrupamento combina três diferentes técnicas para a mineração de dados a partir do código: distância de compressão normalizada (NCD - *Normalized Compression Distance*) Cilibrasi e Vitanyi (2005), Junção de Vizinhos (do inglês *Neighbor Joining* ou NJ) Felsenstein (2003), e um algoritmo de agrupamento baseado em ambiguidade forte. Antes da aplicação do processo de agrupamento propriamente dito, existe uma etapa de pré-processamento, na qual o código fonte deve ser transformado em uma representação simbólica, chamada DNA do programa (Sanches e Cardoso, 2008)(Sanches e Cardoso, 2010)(Sanches et al, 2011). O uso da codificação em DNA visa identificar e destacar as principais estruturas do código fonte, tais como *loops*, operações aritméticas e outras construções de programação, que podem estar relacionadas com otimizações específicas. Dessa forma, a utilização dessa representação simbólica auxilia na identificação das similaridades, realizada pelo algoritmo de agrupamento.

1.1 Motivação

A indústria de circuitos integrados tem conseguido aumentar a densidade de transistores em um único *chip*. Esse crescente avanço tem propiciado a miniaturização dos circuitos, o aumento da capacidade de processamento e de armazenamento do *hardware*, bem como a redução no seu custo de produção. Esses fatores permitem a disponibilização de *chips* de diferentes tipos e valores, viabilizando a utilização de sistemas embarcados em aplicações com os mais variados níveis de complexidade e das mais diversas áreas, tornando-os cada vez mais presentes em nosso cotidiano. Em 2000, cerca de 80% dos processadores produzidos eram empregados em sistemas embarcados (Tennenhouse, 2000) e, desde então, esse percentual vem crescendo a cada ano. Atualmente, os sistemas embarcados compõem o setor de maior crescimento do mercado de informática, sendo encontrado em dispositivos como: fornos de microondas, televisores, videogames, microondas, máquinas de lavar, telefones celulares, cartões eletrônicos (*smart card*), GPS (*Global Positioning System*), PDAs (*Personal Digital Assistant*), *tablets*, entre outros (Patterson e Hennessy, 2003) (Tanenbaum e Austin, 2013).

O uso direto dos algoritmos tradicionais geralmente não é adequado para os sistemas embarcados, dado que eles possuem limitações/restrições inexistentes nas arquiteturas em que esses algoritmos foram concebidos (Ganssle, 2012). Portanto, a adaptação desses algoritmos à arquitetura adotada é necessária, de modo que seus recursos sejam aproveitados com maior eficiência. Nesse sentido, os códigos das aplicações devem ser otimizados durante a compilação visando obter o melhor desempenho, de acordo com as métricas adotadas, para a arquitetura e tipo de processador escolhidos. Esse processo, envolve a aplicação de diferentes transformações, ou passos de otimização, sobre o código fonte.

Os compiladores modernos possuem uma grande quantidade de otimizações que interagem entre si e com o código de entrada de um modo complexo (Kulkarni e Cavazos, 2012). Tradicionalmente, compiladores padrões (ex: GCC) utilizam níveis de otimização que são selecionáveis através de um conjunto simples de parâmetros. Os passos de otimização adotados em cada nível são escolhidos visando melhorar o desempenho de um conjunto representativo de programas, não considerando as características/propriedades específicas de cada código. Entretanto, diversos autores têm demonstrado que a seleção dos passos de otimizações e sua ordem de aplicação (sequência de otimização) podem ter um impacto significativo no desempenho, além de serem dependentes da plataforma e da aplicação (Almagor et al, 2004)(Hoste e Eeckhout, 2008)(Jantz, 2010). Portanto, não existe uma solução única que produza a melhor otimização para todos os códigos (Kulkarni, 2007).

Uma forma de garantir a obtenção da melhor sequência de otimização é testar todas as K possibilidades de combinações que podem ser geradas a partir de uma lista de N

passos de otimização ($K \gg N$). Apesar de simples, essa abordagem é usualmente inviável devido ao tamanho do espaço de busca. Portanto, os desenvolvedores tipicamente usam seus conhecimentos (*expertise*) em um processo intenso e trabalhoso de modificação do código fonte e teste de várias alternativas de compilação, visando obter sequências de otimização satisfatórias para uma função ou aplicativo específico. Nesse contexto, um esquema de DSE é desejável para encontrar a sequência de otimização que resulte no melhor desempenho, segundo a métrica adotada (ex: tempo de execução, tamanho do código, consumo de energia, etc.), para um dado fragmento de código (ou função), considerando o processador alvo e o conjunto de otimizações suportados pelo compilador. Entretanto, os requisitos rigorosos de *time-to-market* comumente existentes nos projetos de sistemas embarcados impõem restrições quanto ao tempo de execução do DSE, sendo, portanto, fundamental acelerar o processo de exploração.

Neste contexto, nós empregamos técnicas híbridas de mineração de dados para explorar de forma rápida e eficiente o espaço de projeto com o objetivo de encontrar sequências de otimização capazes de melhorar o desempenho do código de acordo com o compilador alvo. Nosso trabalho foca na redução do espaço de busca e do tempo de exploração necessários, sem comprometer significativamente o desempenho alcançado pelas sequências de otimização resultantes. Embora nossa abordagem possa ser aplicada para melhorar o desempenho de qualquer tipo de sistema computacional, ela é especialmente importante para a área de sistemas embarcados.

Durante nossa pesquisa foram consideradas as seguintes premissas:

- Existe uma grande demanda por sistemas de computação otimizados e de alto desempenho, tanto para suprir aplicações existentes, quanto para viabilizar outras cuja tecnologia atual não permite. Isso é particularmente observado na área de sistemas embarcados, a qual tem apresentado um grande crescimento e intensas atividades de pesquisa e desenvolvimento.
- Apesar de aceleradores em *hardware* serem melhores para aumentar o desempenho das seções e funcionalidades críticas dos programas legados, boa parte da lógica do sistema ainda é mantida em *software*. Portanto, uma otimização personalizada desses códigos durante a compilação pode ser importante para soluções de alto desempenho na área de sistemas embarcados, os quais geralmente possuem recursos restritos e requisitos rígidos de projeto.
- Os compiladores modernos são atualizados constantemente. Isso envolve a inclusão de novas otimizações, dificultando o aprimoramento e/ou especialização das regras de seleção e sequenciamento dos passos de otimização comumente usadas para melhorar o desempenho do código. Portanto, existe uma necessidade premente para o desenvolvimento de novas abordagens capazes de encontrar a melhor ordem

de aplicação dos passos de otimização de acordo com o código manipulado e o processador alvo, principalmente em aplicações dos domínios de alto desempenho e sistemas embarcados.

1.2 Objetivos

Nosso principal objetivo é desenvolver um método capaz de explorar de forma rápida e eficiente boas sequências de otimização visando obter o melhor desempenho para as funções compiladas, de acordo com o processador alvo e a função de avaliação (métrica objetivo) adotada. Nesse contexto, nosso estudo evolui em duas frentes de pesquisa: a implementação de uma abordagem para a seleção de passos de otimização baseada em técnicas híbridas de mineração de dados complexos, e de novas estratégias de DSE capazes de utilizar os passos selecionados para melhorar o desempenho da busca pelas melhores sequências. Essas abordagens trabalham de forma integrada e foram concebidas de modo a atender os seguintes objetivos específicos:

- Abstrair do processo de seleção de passos de otimização a necessidade de qualquer conhecimento explícito sobre os passos de otimização disponíveis, sua ordem ou a etapa de sua aplicação pelo compilador, bem como da relação entre esses passos e as propriedades dos códigos a serem compilados. Dessa forma, nosso método não requer qualquer estudo prévio sobre como a iteração dos passos afeta a eficiência da otimização, ou treinamento de um modelo capaz de mapear essas relações. Todo o conhecimento utilizado no processo de seleção é extraído implicitamente a partir do histórico de compilações (experiências prévias).
- Determinar os passos de otimização possivelmente relevantes ao problema a partir de um histórico de compilação (conjunto de referência), formado por códigos/funções já compiladas. Nesse processo, a escolha é baseada apenas na semelhança da nova função em relação àquelas presentes no conjunto de referência (decisão por similaridade de padrões no código).
- Restringir as regiões exploráveis do espaço de soluções a partir da seleção prévia de potenciais otimizações, tornando o espaço de busca mais gerenciável.
- Adotar métodos simples e determinísticos de busca que proporcionem um melhor compromisso entre a qualidade das sequências e o tempo de exploração, ou seja, métodos capazes de reduzir o custo computacional da exploração, mantendo os valores alcançados para a métrica objetivo próximos com aqueles resultantes de métodos heurísticos mais complexos (ex: algoritmos genéticos).

- Definir sequências específicas ao problema, ou seja, durante a exploração, a abordagem deve considerar as características e propriedades do código e do processador almejados, de modo a alcançar os valores ótimos (ou próximo disto) para a métrica de avaliação considerada.
- Viabilizar uma exploração significativamente mais rápida sem comprometer a qualidade dos resultados alcançados.

Além disso, neste trabalho também foram analisadas algumas características do método proposto com o objetivo de:

- Verificar o efeito do agrupamento adotado sobre desempenho do método de seleção.
- Avaliar a importância da forma de representação do código usada como entrada no desempenho do método.
- Investigar o impacto da matriz de similaridade na abordagem baseada em agrupamento, usando diferentes tipos de compressores e outras métricas para aferir a similaridade a partir dos códigos DNA.
- Investigar a influência da codificação do DNA na qualidade dos agrupamentos resultantes e, conseqüentemente, das sequências geradas.
- Analisar diferentes estratégias para selecionar sistematicamente o conjunto de referência (histórico de funções) e avaliar o impacto dessas estratégias na eficiência do método de exploração.

1.3 Organização do Texto

O restante deste documento está dividido em 6 capítulos, como segue:

- **Capítulo 2:** apresenta um breve embasamento teórico sobre compilação e otimização, introduzindo uma visão geral de compiladores, sua estrutura típica e uma revisão sobre compiladores otimizantes. O problema de ordenação dos passos de otimização é definido formalmente e são discutidos alguns aspectos teóricos e motivacionais sobre o mesmo.
- **Capítulo 3:** aborda o processo de exploração do espaço de projeto (DSE), apresentando alguns conceitos básicos e descrevendo brevemente os métodos comumente usados na exploração. Ao final, um resumo sobre trabalhos correlatos que investigaram abordagens de DSE voltadas à ordenação dos passos de otimização é apresentado.

- **Capítulo 4:** descreve nossa abordagem para a seleção dos passos de otimização baseada em agrupamento, introduzindo os fundamentos das técnicas para representar as principais estruturas do código fonte, e para extrair os padrões escondidos (mineração de dados) a partir dessa representação.
- **Capítulo 5:** apresenta nossas estratégias de DSE para as sequências de otimização, detalhando os algoritmos usados e como eles utilizam o conjunto de otimizações selecionado, bem como descreve o ambiente de compilação usado para desenvolver essas estratégias de exploração.
- **Capítulo 6:** descreve os experimentos realizados a fim de validar a eficiência e efetividade de nossa abordagem, bem como investigar o impacto de variações nas configurações e entradas das técnicas na acurácia dos resultados alcançados.
- **Capítulo 7:** apresenta nossas conclusões e contribuições, lista as publicações resultantes desta tese e sugere novas frentes de trabalho relacionados ao tema que poderão ser pesquisadas em trabalhos futuros.

Capítulo 2

Compilação e Otimização

Nesse capítulo é apresentada uma visão geral da estrutura de um compilador, dando um enfoque especial ao processo de otimização e à necessidade de se definir uma sequência adequada de otimizações. O problema da ordenação dos passos de otimização é definido formalmente, sendo que aspectos teóricos e motivacionais acerca do mesmo são discutidos.

2.1 Compiladores

Um compilador pode ser definido como um programa de computador que converte um código escrito em uma linguagem de programação, também conhecido como código fonte, em outro código semanticamente equivalente (que faz a mesma coisa e gera o mesmo resultado). O termo "compilador" é usualmente empregado para designar programas que traduzem o código fonte de uma linguagem de programação de alto nível para um código objeto escrito em uma linguagem de programação de baixo nível (ex: linguagem de montagem), específica para o processador que executará o código compilado. Entretanto, existem outros tipos de compiladores que geram códigos em outras linguagens de alto nível ou, até mesmo, versões do código na mesma linguagem de programação original. Esses compiladores são denominados *source-to-source* e visam, dentre outras coisas, otimizar o código gerado segundo algum critério de avaliação (métrica alvo ou objetivo).

Os primeiros compiladores, denominados montadores (*Assemblers*), realizavam traduções diretas das instruções de códigos escritos em linguagem de montagem (ex: "add r3,r1,r2") em instruções reconhecidas e executáveis pelo processador (ex: "0xE0813002" em ARM) (Johnson, 2004). Com o surgimento das linguagens de alto nível, esse processo de tradução ficou mais complexo para lidar com a abstração introduzida através das instruções dessas linguagens, as quais não têm relação direta com as instruções de baixo nível. Os primeiros compiladores de alto nível tipicamente geravam programas que eram significativamente mais lentos que os códigos em linguagem de montagem (*Assembly*) escritos

manualmente, dificultando sua adoção em larga escala (Kulkarni, 2007). Essa deficiência impulsionou a indústria de compiladores a investir em técnicas capazes de melhorar a qualidade do código gerado na compilação. Isso resultou no desenvolvimento de inúmeras técnicas de otimização destinadas à análise do código fonte e sua tradução em um versão mais eficiente (ex: mais rápida ou menor) sem afetar sua funcionalidade original, ou seja, versões semanticamente similares. Com a expansão das funcionalidades inerentes às linguagens de programação mais recentes e com a complexidade crescente das arquiteturas de computadores, os compiladores tornaram-se cada vez mais complexos.

2.1.1 Estrutura básica do compilador

A transformação de um programa escrito em uma linguagem de programação de alto nível em um programa equivalente em linguagem de máquina é realizada pelo compilador. Um compilador também pode realizar outros tipos de transformações como traduzir um código objeto de uma arquitetura para outra, ou mesmo traduzir um código fonte de um linguagem para outra.

Para que essa transformação seja realizada, o processo de compilação é dividido em etapas que analisam um dado formato e o sintetizam em um novo, tipicamente partindo-se de uma sequência de caracteres do código fonte e gerando o código objeto. As etapas principais da compilação são:

1. **Análise léxica:** analisa o texto apresentado e o divide em pequenos pedaços, chamados *tokens*, os quais podem ser membros válidos da linguagem em que o código foi escrito. Nessa etapa, o compilador verifica a existência de inconsistências, ou seja, quando alguma parte do texto não pode ser convertida em um *token* válido.
2. **Análise sintática:** processa os *tokens* e gera uma representação intermediária (em inglês, *Intermediate Representation* ou IR), que pode ser uma representação sintática sequencial ou em árvore abstrata (AST - *Abstract Syntax Tree*), além de uma tabela de símbolos formada pelos identificadores usados no programa e seus atributos. Nesse processo, o compilador verifica erros de sintaxe, os quais ocorrem quando alguma sequência de *tokens* não é reconhecida.
3. **Análise semântica:** processa a representação intermediária (IR) para verificar se o programa atende à semântica exigida pela linguagem de origem. Por exemplo, a verificação de todos os identificadores usados e suas respectivas declarações com os tipos compatíveis. Durante essa análise, o compilador verifica se o programa atende a todos os requisitos de semântica da linguagem usada.
4. **Geração de código:** transforma a representação intermediária em um código de máquina equivalente, o qual pode estar na forma de um módulo realocável ou diretamente em um programa executável.

Além dos componentes que realizam as quatro etapas principais descritas, a tabela de símbolos e de acesso à rotinas e uma interface com o sistema operacional também fazem parte do compilador, conforme demonstrado na Figura 2.1. A interface é usada para realizar a leitura e gravação de arquivos, para fazer a comunicação com o usuário e para facilitar a portabilidade de um compilador entre sistemas operacionais.

Figura 2.1: Estrutura em alto nível de um compilador simples (Muchnick, 1997).

A estrutura tradicional de um compilador é composta por dois módulos: *front-end* e *back-end*. O *front-end* compreende as fases iniciais do processo, responsáveis pela geração da representação intermediária (IR) a partir do código fonte. Representações intermediárias, como por exemplo código de três endereços e grafos de fluxo de dados e controle (CDFG - *Control Data Flow Graph*), estão mais próxima do código alvo do que o código fonte, mas ainda permite uma manipulação mais fácil do que se fosse utilizado o código objeto.

O *back-end* envolve as fases finais da compilação, que recebem essa IR e geram o código objeto pela arquitetura alvo. Essa estruturação facilita a construção de compiladores para linguagens e plataformas diferentes. Por exemplo, a substituição de um *front-end* baseado em uma linguagem de programação por outro, que suporte uma linguagem de programação diferente, é mais fácil com essa estrutura. De forma similar, para gerar códigos-objeto para arquitetura diferentes daquela inicialmente projetada, basta substituir o *back-end*. Em alguns casos, o compilador pode possuir um módulo intermediário chamado *middle-end*, responsável por realizar transformações na IR, de modo a produzir um código objeto mais eficiente.

2.1.2 Otimizações

A Otimização de código é a estratégia de examinar a representação intermediária produzida pelo *front-end* do compilador com o objetivo de produzir, através de algumas transformações ou passos de compilação, um código mais eficiente. O termo "otimizador" deve ser entendido de maneira informal, pois não é possível garantir que, ao ler um programa P e gerar outro P' equivalente, o otimizador sempre atingirá a melhor eficiência de P' segundo a métrica de desempenho adotada, o que estaria de acordo com a noção formal de otimização (Muchnick, 1997). De fato, a solução ótima no sentido formal, é raramente encontrada em uma escala global simplesmente devido à complexidade inerente da análise. No escopo de compilação, a otimização é composta por duas etapas: análise e transformação. Enquanto a análise deve identificar oportunidades de modificações a serem realizadas, sejam elas aplicadas nas instruções, nas variáveis ou nas estruturas do código, a transformação deve efetuar essas mudanças como um resultado direto da análise (Johnson, 2004).

Várias transformações se reúnem sob o nome de otimização. Esses passos de otimização visam analisar uma representação do programa (intermediária ou de baixo nível) e transformá-la, de modo que ela realize a mesma tarefa de forma mais eficiente. Dependendo das características de cada aplicação, pode-se tirar maior ou menor proveito de cada tipo de transformação. Uma das técnicas de otimização mais eficazes e independente de arquitetura é a otimização de laços (*loops*), pois esse tipo de estrutura costuma ser um bom candidato para melhorias. Além das otimizações que atuam nos laços, alocação de registradores e escalonamento de instruções também são outros exemplos de transformações relevantes.

As otimizações podem ocorrer em diferentes etapas da compilação (Muchnick, 1997): (i) aplicadas diretamente ao código fonte ou alguma representação intermediária de alto nível que preserve a estrutura geral do programa; (ii) aplicadas à representação intermediária de nível alto ou médio; independentes da arquitetura, (iii) aplicadas em código de baixo nível que necessitam de informações dependentes da arquitetura; (iv) aplicadas em tempo de ligação (*linking*) que operam no código objeto realocável.

Quando as otimizações ocorrem em uma representação intermediária de nível alto ou médio (*middle-end*), elas não consideram os detalhes específicos da arquitetura, enquanto se elas ocorrem em uma representação de baixo nível (*back-end*), elas são fortemente ligadas à arquitetura alvo. Dessa forma, as otimizações implementadas no *middle-end* podem ser facilmente portadas de uma arquitetura para outra (independência da plataforma), assim como podem ser aplicadas em representações intermediárias (IR) geradas a partir de códigos escritos em diferentes linguagens de programação (independência da linguagem). Esse tipo de transformação é particularmente adequado para compiladores multi-plataformas e multi-linguagens, cuja estrutura típica é apresentada na Figura 2.2 (Quéva,

2007). Em contrapartida, as otimizações em baixo nível exploram mais as características da plataforma computacional, como os modos de endereçamento suportados pela arquitetura (Menotti, 2010).

Figura 2.2: Estrutura típica de um compilador multi-plataforma e multi-linguagens.

As otimizações são interessantes de serem aplicadas em uma representação intermediária principalmente pelas seguintes razões (Johnson, 2004):

- As instruções em IR são mais simples que as instruções no código fonte, simplificando a análise.
- A representação intermediária tem um efeito normalizador sobre os programas. Diferentes códigos fonte produzem códigos em IR similares.
- A IR tende a ser uniforme sobre diversas arquiteturas alvo. Assim, um mesmo algoritmo de otimização pode ser aplicado a diferentes plataformas.

2.1.3 Compiladores otimizantes

A estrutura clássica dos compiladores otimizantes que dominaram a área desde os anos 60 até o final dos anos 90 pode ser expressada pela Figura 2.3 (Cooper et al, 2002). Nessa estrutura, o compilador consiste de uma sequência fixa de passos aplicados em

Figura 2.3: Estrutura de um compilador otimizante clássico.

uma ordem pré-selecionada. O compilador otimizante clássico aplica a mesma sequência de passos em todos os códigos compilados.

Alguns ambientes de compilação recentes utilizam uma estrutura mais adaptativa, proposta originalmente por Cooper et al (2002), na qual o otimizador de ordem fixa é substituído por um *pool* de transformações que podem ser ativadas (ou não) em uma ordem definida especificamente para o código a ser compilado. Esse processo incorpora dois novos componentes à estrutura do compilador: um mecanismo decisor, responsável por selecionar o conjunto de operações a serem empregadas e a ordem em que as mesmas devem ser aplicadas sobre o código; e uma função objetivo, a qual é usada sobre código de máquina, retornando as avaliações do código de acordo com as métricas de desempenho adotadas. A saída da função objetivo realimenta o mecanismo decisor de modo a refinar a sequência de otimizações, permitindo assim que esse mecanismo explore o espaço de possíveis sequências de otimização. A Figura 2.4 ilustra esse processo, também conhecido por compilação iterativa (Cooper et al, 2002).

Figura 2.4: Estrutura de um compilador otimizante adaptativo.

A abordagem investigada no presente trabalho se insere no ramo da compilação iterativa e se assemelha à estrutura proposta por Cooper et al (2002), com algumas especifici-

dades a serem detalhadas no Capítulo 5.

2.1.4 Compiladores e sistemas embarcados

Um sistema embarcado pode ser considerado um sistema computacional com propósitos específicos, normalmente construído em dimensões reduzidas e que deve funcionar de forma autônoma. Tais sistemas geralmente possuem restrições de projeto e a eficácia das soluções concebidas exerce uma importante influência na sua viabilidade. Para conceber uma solução de alto desempenho, cuja resposta seja adequada aos requisitos impostos no projeto, deve-se buscar abordagens de desenvolvimento que propiciem o aumento do poder computacional disponível.

Cardoso e Diniz (2008) apresentam o fluxo de desenvolvimento de um típico projeto de SoC (*System on Chip*) em FPGA, dando maior ênfase nos processos de compilação e síntese do *hardware* reconfigurável, conforme pode ser observado na Figura 2.5.

Figura 2.5: Estrutura de um compilador para *hardware* reconfigurável (Cardoso e Diniz, 2008).

Como no compilador tradicional, o *front-end* é responsável por reconhecer os programas aceitos (e rejeitados), informar os erros de sintaxe e semânticos, e gerar a representação intermediária (IR) e o mapeamento de armazenamento preliminar. Nessa fase, estruturas adicionais podem ser incorporadas no código fonte a fim de ressaltar sua natureza concorrente ou sequencial, enquanto mantém a semântica da linguagem de entrada. Por exemplo, para linguagens com ordem de execução não-determinista, o *front-end* pode impor uma sequência específica ou gerar uma representação concorrente (Cardoso e Diniz, 2008).

No *middle-end* é realizado o aprimoramento/otimização do código, visando atingir aos requisitos do projeto (tempo de execução, área, consumo de potência, etc.). Nessa fase, é aplicada uma série de transformações sobre a IR para incorporar tipos de dados e operações especializadas e explorar o paralelismo inerente à aplicação, sem afetar sua semântica. Essas transformações podem ser tanto independentes da arquitetura alvo (ex: eliminação de subexpressões, redundância e códigos inúteis), quanto direcionadas à arquitetura (ex: transformações em *loops* e transferências de computação para regiões com menor uso). Enquanto algumas dessas transformações exploram aspectos arquiteturais de alto nível, como a existência de múltiplas memórias para aumentar a disponibilidade de dados, outras transformações exploram a capacidade de personalizar unidades funcionais específicas em uma unidade de processamento reconfigurável (RPU), assim, implementando diretamente instruções de alto nível no nível do *hardware*. Por exemplo, a personalização das unidades aritméticas para apoiar diretamente as operações de ponto fixo sobre tipos de dados com tamanhos fora do padrão. As transformações aplicadas no *middle-end* são independentes da plataforma alvo.

O *back-end* de um compilador traduz a representação intermediária para o código de máquina da plataforma alvo, assegurando a conformidade com as interfaces do sistema e as características do projeto. Nesse processo, o compilador realiza a seleção de instruções, a alocação de registradores e o escalonamento das operações. A seleção da instrução envolve a escolha das instruções capazes de implementar as operações da IR. Essa tarefa usualmente é vista como um problema de casamento de padrões e busca produzir, na medida do possível, um código compacto e rápido, capaz de aproveitar as vantagens do dispositivo alvo (ex: uso eficiente dos modos de endereçamento). A alocação de registradores decide qual valor será mantido nos registradores durante a execução do programa. Devido à limitação dos recursos de armazenamento, a alocação ótima é um problema NP-completo e pode ocasionar mudanças nas instruções escolhidas, bem como o uso de comandos adicionais de leitura e gravação na memória (Cardoso e Diniz, 2008). O escalonamento de instruções busca evitar *stalls* e *interlocks* no *hardware* através da utilização eficiente dos elementos lógicos (ex: unidades funcionais) disponíveis. Essa tarefa também pode aumentar o tempo de vida das variáveis através de mudanças nas alocações dos registradores. Normalmente, ela emprega técnicas heurísticas, uma vez que na

maioria dos casos é um problema NP-completo (Cardoso e Diniz, 2008). Nesse contexto, a síntese de alto nível (HLS) é responsável pelo escalonamento das macro-operações e instruções, bem como pela geração da definição de uma arquitetura concreta pela síntese de seus caminhos de dados e unidades funcionais (geração da lógica RTL); a síntese lógica é responsável pela distribuição dessa lógica RTL para os elementos lógicos disponíveis; e o P&R (*place and route*) pelo posicionamento e roteamento físico desses elementos dentro da plataforma. Alguns desses passos podem ser desenvolvidos pela integração de ferramentas de síntese comerciais no fluxo de compilação e, em muitos casos, usando bibliotecas de componentes específicas (IPs) da arquitetura alvo. Entretanto, a automação total dessa etapa ainda é um desafio (Menotti, 2010).

Dentre os diversos processos que compõem o fluxo de compilação, nesta pesquisa estamos particularmente interessados nas transformações de código que ocorrem comumente no *middle-end*, mais especificamente com o problema de ordenação dos passos de otimização, também conhecido como sequenciamento das otimizações. A ordenação dos passos de otimização do compilador é um problema de interesse de pesquisa há várias décadas, sendo de particular relevância para o domínio de aplicações que visam desempenho e com restrições de custo em sistemas embarcados. A variação na ordem de execução desses passos, muitas vezes gera códigos de saída distintos, com características diferentes de velocidade, tamanho de código e consumo de energia. A maioria das abordagens atuais para endereçar esta questão foca no desenvolvimento de métodos inovadores para avaliar seletivamente o vasto espaço de busca das sequências de otimização para produzir uma boa representação (mas, potencialmente sub-ótima) para cada programa (Jantz e Kulkarni, 2013).

Embora no escopo desta tese tenha sido na otimização de funções para serem executadas em processadores embarcados, o fato das otimizações serem realizadas principalmente no *middle-end* permite que nossa metodologia seja facilmente estendida para um uso mais amplo em projetos de SoCs, incluindo, mas não limitando, à otimização dos códigos que serão sintetizados em *hardware*.

2.2 Problema da Ordenação dos Passos de Otimização

À medida que mais operações de otimização começaram a ser implementadas em vários compiladores, logo se percebeu que muitos passos de otimização interagem entre si. Cada passo de otimização tenta aplicar uma série de transformações que consistem em uma sequência de alterações a fim de melhorar a eficiência do código, sem afetar o seu comportamento semântico (Kulkarni, 2007). Muitas dessas otimizações usam e compartilham recursos (como os registradores) e também precisam de condições específicas no código para serem aplicáveis. Como resultado, os passos de otimizações interagem uns com os outros, através da habilitação e desabilitação de oportunidades para outras

otimizações serem aplicadas. Tais interações entre os passos de otimização têm sido amplamente estudadas no contexto de diferentes compiladores, cada qual com seu conjunto distinto de passos de otimização, e variadas arquiteturas (Benitez e Davidson, 1988)(Almagor et al, 2004)(Agakov et al, 2006)(Kulkarni et al, 2007)(Pan e Eigenmann, 2008)(Kulkarni et al, 2010)(Huang et al, 2013)(Jantz e Kulkarni, 2013). Com base nesses estudos, agora é definitivamente conhecido que a interação entre as otimizações faz com que diferentes ordens na aplicação dos passos de otimização possam gerar códigos distintos, com uma variação de desempenho potencialmente significativa entre eles.

Consideremos o seguinte exemplo apresentado em Johnson (2004) para ilustrar o efeito causado pela mudança na ordem de aplicação dos passos de otimização por um compilador, no qual duas transformações podem ser aplicadas: *alocação mínima de registradores* e *escalonamento de instruções*. Suponha um fragmento de código escrito em C com apenas 2 atribuições:

```
a = b;  
c = d;
```

A execução dessas atribuições provoca duas leituras (das variáveis *b* e *d*) e duas escritas (para as variáveis *a* e *c*). Se a transformação *alocação mínima de registradores* for aplicada primeiro, teremos como resultado o seguinte código em linguagem de montagem:

```
ld r1, b  
st r1, a  
ld r1, d  
st r1, c
```

O código resultante empregou apenas um registrador para as operações de leitura e gravação. O problema é que existe uma dependência de dados entre as instruções que torna a aplicação da transformação *escalonamento de instruções* sem efeito, não produzindo mudanças no código. Em um típico processador com estágios de *pipeline*, essas dependências irão resultar em *stalls* de *pipeline* e, conseqüentemente, na redução do *throughput*.

Entretanto, se aplicarmos as transformações na ordem inversa, executando primeiro o *escalonamento de instruções*, teremos o seguinte código *Assembly*:

```
ld r1, b  
ld r2, d  
st r1, a  
st r2, c
```

Nesse caso, dois registradores (*r1* e *r2*) foram utilizados para permitir a independências entre as operações do *pipeline*, possibilitando a execução mais rápida do código. Entretanto, o uso de um número maior de registradores pode forçar a introdução de *spill codes*

pelo alocador de registradores em outras partes do programa, caso não existam registradores suficientes nesse ponto da execução do programa. *Spill codes* são instruções de escrita e leitura inseridas pelo compilador que transferem o conteúdo de registradores para a memória, e então os recarrega quando o programa precisa usar valores previamente transferidos. Isso acarreta dois efeitos indesejáveis: o aumento no tamanho do programa em decorrência dessas instruções extras, e o aumento da comunicação com a memória, provocando uma redução no tempo de execução.

Um requisito fundamental para se encontrar a sequência de otimização ideal é a capacidade do compilador em modificar a ordem de aplicação dos passos de otimização. Muitos compiladores foram projetados sem essa característica. Nesses casos, os formatos aceitos como entrada em cada passo de otimização estão geralmente associados aos formatos de saída de seus antecessores. Na prática, essa dependência entre os passos serializa e fixa a ordem com a qual eles serão aplicados pelo compilador. Com a conscientização sobre os potenciais benefícios da aplicação das transformações em diferentes ordens, muitos projetos de compiladores passaram a adotar uma nova estratégia de compilação, na qual os passos de otimização são aplicados nas representações intermediárias do código (Auslander e Hopkins, 1982)(Benitez e Davidson, 1988). Assim, cada passo de otimização pode ser aplicado repetidamente e em qualquer ordem, resolvendo muitas questões simples relacionadas à ordenação.

Encontrar a melhor ordem para aplicar os passos de otimização é uma tarefa importante em áreas de aplicação nas quais o desempenho é fundamental, tais como sistemas embarcados e computação de alto desempenho, de forma que o código mais eficiente pode ser gerado para cada aplicação. Essa tarefa é comumente conhecida como o problema de ordenação de passos de otimização. Pesquisas anteriores sobre a ordenação de passos de otimização mostraram que não existe uma ordem universal de otimizações capaz de produzir o código ideal para todas aplicações (Cooper et al, 2002)(Cooper et al, 2006)(Kulkarni e Cavazos, 2012). Ao contrário, a melhor ordem depende: (i) do programa a ser otimizado, (ii) da maneira pela qual os passos de otimização são implementados no compilador, e (iii) das características da plataforma alvo (Kulkarni, 2007).

Nas próximas seções, uma definição formal do problema é apresentada com uma subsequente discussão de aspectos teóricos relacionados à decidibilidade do mesmo e à intratabilidade da exploração completa do espaço de busca.

2.2.1 Formalização do problema e aspectos teóricos relacionados

Touati e Barthou (2006) apresentaram um modelo teórico do problema de ordenação de passos de otimização (OPO) que é também discutido em Quéva (2007). A seguir, apresentaremos essa formalização, bem como as principais conclusões a respeito da decidibilidade desse problema e de suas variações também discutidas por Touati e Barthou

(2006).

Seja M um conjunto finito de passos de otimização. O objetivo é construir um algoritmo A com três entradas: um programa P , um conjunto de entradas para esse programa I e uma meta T que deve ser atendida pelo código otimizado, por exemplo, o tempo de execução esperado. Assumimos que existe uma função de avaliação de desempenho t capaz de avaliar ou prever precisamente a métrica de desempenho escolhida. Nesse exemplo, $t(P, I)$ retorna o tempo de execução predito. Para cada código dado como entrada e seu respectivo conjunto de entradas, esse algoritmo A deve computar uma sequência finita de transformações $s = \{m_1, m_2, \dots, m_n\}$, sendo $m_i \in M^*$ que resolva o seguinte problema:

Problema 1 Ordenação de passos de otimização (OPO)

Seja uma função de avaliação de desempenho t previamente especificada, $\forall T \in \mathbb{N}$ (um tempo de execução em ciclos do relógio), $\forall P$ (um programa), $\forall I$ (uma entrada do programa), existe uma sequência $s \in M^$ tal que $t(s(P), I) < T$?*

Em outras palavras, seja o conjunto:

$$S_{t,M}(P, I, T) = \{s \in M^* | t(s(P), I) < T\} \quad (2.1)$$

É possível verificar se o conjunto $S_{t,M}(P, I, T)$ é vazio?

Textualmente, o problema da ordenação de passos de otimização tenta determinar para cada programa e entrada se existe ou não uma sequência de otimizações s capaz de gerar um código que satisfaça a meta T .

Embora essa seja a formulação mais geral do problema, Touati e Barthou (2006) argumentam que a resposta ao **Problema 1** da forma como foi definido depende do desempenho do modelo de predição t . Uma vez que a função t não é definida previamente, o **Problema 1** não pode ser respondido e os autores apresentam uma pequena modificação do mesmo:

Problema 2 (Ordenação de passos de otimização modificada)

Para qualquer função de avaliação de desempenho t , $\forall T \in \mathbb{N}$ (um tempo de execução em ciclos do relógio), $\forall P$ (um programa), $\forall I$ (uma entrada do programa), existe uma sequência $s \in M^$ tal que $t(s(P), I) < T$?*

Em outras palavras, seja o conjunto:

$$S_M(t, P, I, T) = \{s \in M^* | t(s(P), I) < T\} \quad (2.2)$$

É possível verificar se o conjunto $S_M(t, P, I, T)$ é vazio?

Enquanto no **Problema 1** t é considerado um modelo aproximado, no **Problema 2**, t é obtido a partir da execução do código otimizado na plataforma almejada. Ou seja, na formulação do **Problema 2** considera-se o caso em que a arquitetura alvo é conhecida e t é o modelo de avaliação de desempenho mais preciso possível dessa arquitetura alvo.

A partir da formulação do **Problema 2**, Touati e Barthou (2006) provaram que o problema da ordenação de passos de otimização, em sua forma geral, é indecidível. A ideia básica da prova apresentada em Touati e Barthou (2006) consiste em uma redução do problema da linguagem vazia no problema OPO. O problema da linguagem vazia (Sipser, 2007) é um problema clássico da teoria da computação que pode ser assim enunciado:

Problema 3 (Linguagem vazia)

Seja L uma Linguagem recursivamente enumerável, é possível verificar se L é vazia?

Em outras palavras:

Se L é uma linguagem aceita por uma máquina de Turing M , é possível verificar se o conjunto de palavras que essa máquina aceita é vazio?

Sabe-se que esse problema é indecidível pela aplicação do teorema de Rice (Sipser, 2007). Uma vez que Touati e Barthou (2006) foram capazes de mostrar que o **Problema 3** se reduz ao **Problema 2**, e sabe-se que o **Problema 3** é indecidível, conclui-se então que o **Problema 2** também é indecidível. Os detalhes dessa prova podem ser encontrados em Touati e Barthou (2006).

Além desse enunciado principal do problema de ordenação de passos, os autores também discutem uma variação do problema a partir da adoção de uma biblioteca de passos de otimização (OPO-BO). Eles mostram que esse problema é indecidível se a aplicação das otimizações existentes na biblioteca puder gerar um número infinito de diferentes programas.

Com o objetivo de simplificar o problema na direção de modelos decidíveis, os autores propuseram outras formulações simplificadas para tratar casos mais específicos de compiladores. Um deles é o problema da ordenação de passos de otimização com custo de compilação (OPO-CC), no qual se considera a existência de uma função discreta de custo que é capaz de retornar uma avaliação em tempo de compilação. Esse custo pode ser: o tempo de compilação, o número de programas gerados, o tamanho da sequência de otimização, o número de sequências distintas, entre outros. A adição dessa função de custo simplifica bastante o problema da ordenação uma vez que ela é uma função de custo estritamente crescente, o que torna possível que o algoritmo A faça a busca trivial por todas as sequências de otimização com uma limitação no custo, e pode então escolher a melhor sequência dentre elas. Essa abordagem é comumente utilizada em compilação iterativa (Kisuki et al, 1999)(Quéva, 2007). Nesse tipo de compilação, diferentes sequências de otimização são aplicadas (compilação) e avaliadas iterativamente com o objetivo de

alcançar o melhor desempenho para uma aplicação. Como exemplos desse tipo de abordagem podemos citar os compiladores usados nos trabalhos (Cooper et al, 2006)(Kulkarni e Cavazos, 2012)(Jantz e Kulkarni, 2013)(Cardoso et al, 2013).

Outras formulações do problema também tiveram sua decidibilidade investigada e podem ser encontradas em (Touati e Barthou, 2006). A Figura 2.6 apresenta uma síntese da análise de decidibilidade, considerando-se as três formulações citadas no presente trabalho.

Figura 2.6: Visão geral da decidibilidade do problema de ordenação.

Quéva (2007) ressaltou que embora a modelagem do problema e a análise de decidibilidade realizada por Touati e Barthou (2006) possa ser vista como um importante passo para o entendimento do problema de ordenação, esse modelo ainda é vago a respeito da sua decidibilidade quando é usada uma função de avaliação de desempenho que não requer a execução do programa. Como restrição, a função de avaliação t no modelo original não pode ser uma aproximação, e como tal modelo não considera a plataforma utilizada na avaliação de desempenho, apenas o tempo de execução pode satisfazer tal condição. Por outro lado, o modelo mostra aos desenvolvedores de compiladores que as simplificações no problema de ordenação de passos de otimização tornam o problema decidível. Por exemplo, a limitação no tamanho máximo da sequência de otimização permite que a obtenção de um código ótimo e eficiente seja decidível.

Kulkarni (2007), por sua vez, salienta que embora Touati e Barthou (2006) tenham provado que encontrar a sequência de otimização ideal seja indecidível nos esquemas gerais de compilação e com uso biblioteca de passos de otimização, a hipótese dessa prova parte do princípio que o conjunto de todos os programas possíveis de serem gerados por diferentes ordenações de fases é infinito. Essa hipótese é razoável desde que otimizações possam ser aplicadas em um número arbitrário de vezes, gerando códigos distintos. Na prática, no entanto, tipicamente os compiladores impõem uma restrição quanto ao número

de vezes que os passos podem ser utilizados numa sequência de otimização. Dessa forma, encontrar a sequência ideal é de fato decidível na maioria dos compiladores atuais.

2.2.2 Espaço de Busca

Um problema que é teoricamente solúvel (decidível) pode não ter uma solução prática; ou seja, pode não haver algoritmo que resolva-o sem precisar de uma quantidade extraordinária de tempo e/ou memória. Outro aspecto teórico importante relacionado ao problema de ordenação de passos de otimização (OPO) refere-se à sua tratabilidade. Isto é, embora o problema de ordenação possa ser decidível, considerando-se as simplificações inerentes aos compiladores reais, ainda assim é importante analisar aspectos de complexidade acerca desse problema. Em sua definição mais genérica, o problema de OPO pode ser classificado como intratável.

A definição do problema de ordenação dos passos de otimização pode ser muito ampla. Qualquer combinação e ordenação possível das otimizações com comprimento ilimitado, incluindo repetições de passos, pode constituir sequências de otimização válidas. Por exemplo, dados três passos de otimização: a , b e c , o espaço de busca será formado por pontos correspondentes a sequências de diferentes tamanhos. Se considerarmos sequências de até 4 passos, teremos um espaço de busca com 121 possíveis pontos, como segue:

Tamanho 0: $\{\}$ (conjunto vazio)

Tamanho 1: $\{a\}, \{b\}, \{c\}$

Tamanho 2: $\{a,a\}, \{a,b\}, \{a,c\}, \{b,a\}, \{b,b\}, \{b,c\}, \{c,a\}, \{c,b\}, \{c,c\}$

Tamanho 3: $\{a,a,a\}, \{a,a,b\}, \{a,a,c\}, \{a,b,a\}, \{a,b,b\}, \dots, \{b,a,a\}, \dots, \{c,a,a\}, \dots, \{c,c,c\}$

Tamanho 4: $\{a,a,a,a\}, \{a,a,a,b\}, \{a,a,a,c\}, \dots, \{c,c,c,c\}$

Para formalizar esse conceito, considere uma sequência S com k passos de otimização. Uma possível solução (configuração) de S é dada pela lista $s = \{s_1, s_2, \dots, s_k\}$, na qual cada elemento s_i assume um dos valores existentes em um conjunto finito de passos de otimizações do compilador V , sendo $V = \{v_1, v_2, \dots, v_p\}$. Nesse cenário, o espaço do problema E_n é formado pelo conjunto de todas as possíveis configurações de S até um tamanho limite n , o qual é dado por:

$$E_n(S) = \{s = \{s_1, s_2, \dots, s_k\} \mid k \leq n \wedge s_i \in V\} \quad (2.3)$$

Essa definição do problema pode, de fato, levar a um espaço de busca imenso. A interação entre os passos de otimização, possibilita a repetição de uma mesma otimização em várias posições de uma sequência. Consequentemente, o espaço de busca das sequências é praticamente ilimitado. Para um conjunto com p passos distintos de otimização que podem ser aplicados sobre o código e uma limitação no tamanho máximo das sequências potenciais igual a n , o espaço de soluções resultante possui uma cardinalidade η igual a:

$$\eta = \sum_{i=0}^n p^i \quad (2.4)$$

Mais especificamente, nos experimentos descritos no Capítulo 6, utilizamos um espaço de busca que é gerado por todas as sequências de otimização com tamanho máximo igual a 16, formadas a partir de um conjunto de 49 transformações diferentes, as quais podem ser repetidas ou não dentro da sequência. Esse espaço é equivalente ao formado por palavras de tamanho até 16 geradas a partir de um vocabulário com 49 símbolos diferentes, sendo a cardinalidade η desse espaço igual a:

$$\eta = \sum_{i=0}^{16} 49^i \quad (2.5)$$

Esse valor compreende uma quantidade acima de 49^{16} (última parcela do somatório). Em termos práticos, se supormos que a avaliação de cada sequência ao ser processada (ou simulada) leva apenas 1 segundo, o tempo total estimado para a avaliação de todas as sequências possíveis demandaria acima de $3,5 \times 10^{19}$ anos de processamento, o que caracteriza a inviabilidade da exploração exaustiva dessas sequências.

Para tornar o problema um pouco mais factível, abordagens baseadas em podas agressivas e/ou na avaliação de apenas uma parte do espaço de soluções têm sido investigadas. Alguns pesquisadores realizam uma exploração exaustiva sobre um pequeno subconjunto do espaço de soluções (sequências de otimização), gerado a partir de podas agressivas, para investigar as características do espaço de otimizações (Almagor et al, 2004)(Kulkarni et al, 2009). Isso exige que severas limitações práticas sejam aplicadas ao espaço original para tornar a exploração possível. Em muitos trabalhos, métodos heurísticos baseados em técnicas tradicionais de inteligência computacional são usados para avaliar apenas uma parte do espaço de otimização (Cooper et al, 1999)(Kulkarni et al, 2007)(Hoste e Eeckhout, 2008)(Martins et al, 2014b). Subida de montanha (*hill climbing*), recozimento simulado (*simulated annealing*) e algoritmos genéticos são exemplos de técnicas que têm sido empregadas como buscas iterativas para se encontrar sequências de otimização melhores do que as aquelas usualmente adotadas nos compiladores. A maioria dos trabalhos relata melhorias de desempenho em relação ao uso de uma sequência fixa do compilador.

Na abordagem aqui investigada, e que será detalhada no Capítulo 4, utilizamos uma combinação de técnicas advindas da área de mineração de dados e métodos de busca para encontrar as melhores soluções para o problema OPO a partir da seleção prévia dos passos de otimização. O desempenho alcançado é comparado com aquele obtido por uma abordagem tradicional baseada em algoritmos genéticos.

Capítulo 3

Exploração das Sequências de Otimização

Considerando a grande quantidade de passos de otimização disponíveis nos compiladores modernos, o tamanho e complexidade dos códigos das aplicações embarcadas, e os requisitos e restrições impostos ao projeto do sistema, encontrar a melhor sequência de otimização do compilador para uma aplicação e processador específicos é uma tarefa extremamente difícil, mas fundamental, em uma ambiente de compilação para sistemas embarcados. Essa exploração deve ser rápida e eficiente a ponto de avaliar diferentes soluções potenciais (sequências de otimização) e obter uma versão otimizada do *software* que, ao ser executada no processador desejado, reduza ao máximo as métricas de desempenho desejadas, tais como o tempo de execução e/ou o consumo de energia, sem comprometer os demais requisitos e restrições do projeto do sistema embarcado. Essa tarefa, conhecida como Exploração do Espaço de Projeto (DSE - *Design Space Exploration*), visa avaliar e comparar diversos pontos de espaço de busca (sequências de otimização). O principal objetivo do DSE é encontrar uma solução que apresente a melhor sinergia entre a aplicação e o processador específicos a fim de atender aos requisitos e restrições do projeto e otimizar o tempo de execução (latência).

O DSE pode ser visto como um processo iterativo com dois problemas ortogonais básicos: a avaliação de um ponto do espaço de projeto, o qual representa uma solução candidata (sequência de otimização); e a exploração do espaço de forma mais abrangente possível (maior cobertura), dado que a busca exaustiva é comumente inviável devido ao tamanho desse espaço, conforme discutido na Seção 2.2.2. A escolha do método de avaliação busca um compromisso entre a precisão e o custo do modelo de avaliação. A seleção do método de busca avalia a granularidade (precisão) da cobertura do espaço de projeto. Essas decisões devem possibilitar a automação do processo de exploração.

Nesse capítulo são introduzidos alguns conceitos básicos referentes ao processo de exploração do espaço de projeto, incluindo a descrição de alguns métodos comumente empregados para viabilizar a busca e avaliação de boas soluções durante a exploração

de grandes espaços de projeto. Ao final, é apresentada uma revisão sobre as principais técnicas já investigadas na literatura para a exploração de sequências de otimização.

3.1 Processo de DSE

A Figura 3.1 ilustra o fluxo dos processos de DSE manual e automático (Silvano et al, 2010). O processo convencional de DSE inicia-se com a definição do espaço de busca do projeto, através do modelamento do problema. Após essa definição, o projetista escolhe uma solução desse espaço, a qual considera satisfazer aos requisitos do problema, que é adotada como uma configuração inicial. A solução escolhida é então avaliada a fim de determinar sua adequação. A partir do resultado obtido, o projetista decide o que deve ser alterado na solução candidata a fim de melhorá-la (adequação da solução), iniciando assim um processo iterativo de exploração. Por ser manual, o processo costuma demandar grande esforço e tempo, principalmente nas etapas mais avançadas do projeto, onde as adequações se tornam mais complexas e difíceis. Além disso, a qualidade da solução encontrada depende da habilidade e experiência prévia do projetista e está mais suscetível a erros.

Figura 3.1: Comparativo entre o fluxo manual e automático de DSE.

O DSE automático envolve as mesmas etapas do processo manual, mas costuma ser muito mais rápido. Esse ganho se deve ao melhor desempenho durante a exploração. No fluxo automático, a solução é um modelo configurável que permite a representação de qualquer ponto do espaço de busca e os critérios de avaliação são combinados com métodos eficientes de busca para direcionar a exploração desse espaço (Silvano et al, 2010). Nesse caso, a qualidade da solução encontrada depende dos métodos de avaliação e exploração adotados. Geralmente, as abordagens de DSE automáticas conseguem obter soluções ótimas, ou próximas dela.

3.2 Espaço de Projeto

O conjunto de soluções potenciais, denominado espaço de projeto ou de busca, é determinado com base na quantidade de soluções que devem ser exploradas e os possíveis valores de desempenhos a elas associados. A avaliação exaustiva de todas as potenciais sequências de otimização exige a resolução de dois sub-problemas, os quais sempre foram considerados inviáveis para os compiladores atuais (Kulkarni, 2007):

- *Enumerar* todas as possíveis sequências de otimização: essa enumeração exaustiva requer a exploração completa do espaço de soluções, o qual cresce exponencialmente em função da quantidade de passos de otimização. Dado que os compiladores modernos implementam um número muito grande de otimizações, tal enumeração é impraticável sem a adoção de alguma forma de restrição do espaço.
- *Avaliar* o desempenho de todas as sequências enumeradas: essa avaliação requer repetidas compilações e execuções do código a fim de encontrar a sequência de otimização que resulta no desempenho ótimo. O custo computacional desse processo costuma ser elevado em relação às demais etapas da exploração. Além disso, muitos ambientes de desenvolvimento de sistemas embarcados não suportam compilação, forçando a adoção de simuladores, o que onera ainda mais o custo da avaliação.

Segundo Gries (2004), a exploração desse espaço pode ser tratado sob dois enfoques distintos: espaço do problema ou espaço de objetivos, sendo que o primeiro é relacionado com a tarefa de enumerar e o segundo com a tarefa de avaliar.

3.2.1 Espaço do problema

O espaço do problema, também conhecido como espaço de decisão ou de soluções, é definido pelos parâmetros (propriedades) de configuração que representam as características naturais do espaço de projeto. Por exemplo, em nosso contexto, o espaço do problema é formado por todas as sequências geradas a partir dos passos de otimização aceitos no ambiente de compilação utilizado. Esses parâmetros são parte da descrição do problema e não têm qualquer indicativo dos valores das métricas de desempenho adotadas na avaliação das soluções. Assim, a exploração baseada nesse espaço consiste na escolha da sequência de otimização, na sua avaliação e na definição de sua adequação ao problema.

O aumento no número de passos de otimização aceitos pelo compilador, no tamanho máximo das sequências, ou na quantidade e possíveis valores dos parâmetros de configuração de cada otimização, eleva significativamente a complexidade do processo de exploração. Diversas escolhas devem ser avaliadas a fim de subsidiar as decisões tomadas durante a investigação do espaço de soluções, selecionando aquelas que apresentam os

melhores valores para a métrica de avaliação (ex: menor latência, maior *speedup*, menor consumo de energia, etc.) e considerando as restrições impostas ao sistema.

A cardinalidade do espaço do problema pode explodir exponencialmente com o aumento arbitrário dos parâmetros de configuração. Essa cardinalidade seria ainda maior se as otimizações permitissem diferentes valores de configuração, como ocorre com os fatores de desdobramento de laços (*loop unrolling*). Portanto, a busca exaustiva pela melhor solução é impraticável, sendo necessária a adoção de uma abordagem exploratória sistemática capaz de caracterizar de forma representativa esse espaço (Gries, 2004).

3.2.2 Espaço de objetivos

O espaço de objetivos é definido pelos valores das métricas de avaliação adotadas pelo DSE, tais como: tempo de execução, tamanho do código gerado, consumo de energia, entre outros. A exploração nesse espaço consiste em restringir sistematicamente as soluções candidatas em relação à métrica alvo.

Em problemas com um único objetivo (mono-objetivo), o espaço de objetivos é totalmente ordenado de acordo com a função custo adotada (Palermo et al, 2005). Portanto, a otimização consiste em buscar o melhor resultado através da minimização ou maximização dessa função.

A exploração baseada em espaços multi-objetivos busca otimizar métricas comumente conflitantes entre si. Logo, uma configuração que melhora um objetivo costuma comprometer outro, de modo que encontrar uma única solução que seja superior em todos eles é geralmente impossível. Nesse caso, ao invés de uma solução única é obtido um conjunto de soluções. A escolha de qual solução deve ser adotada depende exclusivamente da priorização de uma determinada métrica sobre as demais. O processo de avaliação de um ponto dentro do conjunto de soluções envolve sua comparação com os demais pontos. Nesse processo, cada objetivo é comparado individualmente. O aumento na quantidade desses objetivos eleva a dimensionalidade do problema e, conseqüentemente, a complexidade do DSE.

3.3 Redução do Espaço de Busca

Devido à quantidade de transformações disponíveis nos compiladores atuais para os diversos processadores embarcados, se torna inviável avaliar todas as sequências de otimização a fim de encontrar a mais adequada para um determinado código. Mesmo com a utilização de simulações de alto nível e algoritmos de exploração eficientes, o tempo para a avaliação dos pontos do espaço de soluções normalmente representa o gargalo do DSE (Piscitelli e Pimentel, 2012). Portanto, é fundamental adotar estratégias capazes de reduzir eficientemente o número de pontos do espaço (sequências potenciais) que

devem ser explorados (Palermo et al, 2009). Além de limitar a quantidade de pontos visitados durante a busca, também é necessário garantir que as sequências avaliadas sejam suficientemente representativas de modo a não comprometer a qualidade dos resultados alcançados. Gries (2004) agrupa as várias abordagens de redução do espaço de busca nas seguintes categorias:

- **Exploração hierárquica** (Mohanty et al, 2002)(Eyerman et al, 2006): é baseada em níveis hierárquicos de exploração. No nível mais elevado, é realizada uma exploração grosseira do espaço de busca a fim de identificar e classificar (*ranking*) as regiões de interesse do problema. Essas regiões são então submetidas aos métodos dos níveis menores de modo a refinar a exploração. A ideia básica é que quanto menos pontos existem na região em análise, mais detalhado pode ser a exploração. Além disso, a busca pode alternar entre os níveis de exploração, de modo que os modelos do nível superior possam ser realimentados com os resultados das explorações dos níveis inferiores para melhorar a caracterização do espaço (Gries, 2004). A quantidade de níveis hierárquicos pode variar de acordo com a complexidade do espaço de projeto, a capacidade de refinamento de cada nível e seu custo computacional.
- **Amostragem do espaço de projeto** (Cooper et al, 2006)(Hoste e Eeckhout, 2008): a exploração é realizada apenas sobre uma amostra de pontos do espaço de projeto. Apesar de não garantir a existência de soluções ótimas nas amostras, essa abordagem é uma escolha razoável quando se deseja uma exploração não tendenciosa e a busca exaustiva não é viável (Gries, 2004). O padrão de amostragem adotado pode ser totalmente aleatório, baseado em alguma forma de distribuição (ex: uniforme ou gaussiana), ou definido de acordo com a forma esperada do espaço de projeto e/ou da função de avaliação. Por exemplo, buscas baseadas no algoritmo de arrefecimento simulado (*simulated annealing*) e algoritmos evolutivos usam uma amostragem aleatória, enquanto que abordagens baseadas na quantização dos parâmetros adotam um padrão de amostragem regular.
- **Subdivisão do problema** (Givargis et al, 2002): essa categoria visa reduzir o número de possíveis soluções através da divisão do problema em subproblemas independentes. Por exemplo, nosso problema pode ser tratado em duas etapas: seleção dos passos de otimização e ordenação das otimizações selecionadas. Assim, ao invés de considerar todas as alternativas do espaço de projeto original, a exploração lida com os espaços para cada um desses subsistemas (Gries, 2004). Embora a dimensão dos espaços de objetivos seja a mesma para os diferentes subsistemas, seus espaços de soluções são geralmente bem menores, dado que envolvem parâmetros específicos de cada subsistema.
- **Análise de sensibilidade** (Palermo et al, 2009)(Silvano et al, 2010): esse tipo de

abordagem reduz a quantidade de parâmetros manipulados durante a exploração em função de sua influência no projeto. A análise de sensibilidade consiste na realização de várias execuções sobre um conjunto de funções (*benchmark*). Em cada execução, apenas um único parâmetro de configuração tem seu valor alterado, mantendo-se os demais com os mesmos valores. A sensibilidade de um parâmetro para cada função é dada pela diferença entre os valores máximo e mínimo dos resultados obtidos a partir da variação no parâmetro. A sensibilidade global do parâmetro pode ser obtida a partir da média das sensibilidades das funções do conjunto. Os parâmetros são então ordenados de acordo com sua sensibilidade. A exploração inicia-se com a fixação do valor do parâmetros menos sensíveis e variação do parâmetro mais sensível até a otimização da métrica objetivo (Gries, 2004). Então, esse parâmetro é fixado no valor ótimo e o processo recomeça para o segundo parâmetro de maior sensibilidade. Isto ocorre até que todos os parâmetros tenham sido configurados. O principal problema dessa abordagem é que ela assume a independência entre os parâmetros (Ascia et al, 2004), o que não ocorre na prática, principalmente no problema OPO, como já discutido.

- **Restrição do espaço** (Almagor et al, 2004)(Haneda et al, 2005)(Kulkarni, 2007)(Pan e Eigenmann, 2008): essa categoria se baseia na identificação de restrições nos parâmetros de projeto de modo a reduzir a quantidade de pontos do espaço de soluções sem comprometer, pelo menos significativamente, os valores que podem ser atingidos no espaço de objetivos (métrica alvo). Por exemplo, resultados de perfilamento (*profiling*) podem ser empregados para determinar as otimizações mais relevantes para uma dada função.

Neste trabalho, utilizamos uma abordagem baseada em agrupamento para restringir o espaço de busca através da seleção de passos de otimização potencialmente relevantes. Essa abordagem é descrita no Capítulo 4.

3.4 Métodos de Exploração

A eficiência do DSE está relacionada com sua capacidade de cobertura do espaço de projeto em um curto período de tempo (Sigdel, 2011). Para atingir tal objetivo, é necessária a adoção de métodos de busca que permitam explorar diferentes pontos desse espaço e, ao mesmo tempo, sejam capazes de guiar o DSE em direção às melhores soluções. Segundo Gries (2004), esses métodos podem ser agrupados de acordo com a estratégia de exploração adotada, como segue:

- *Busca exaustiva*: é uma estratégia de busca baseada na força bruta que garante a cobertura total do espaço de projeto, ou seja, todos os possíveis pontos do espaço são visitados. Essa abordagem sempre garante a obtenção da melhor solução

global. Entretanto, esse tipo de busca é inviável para grandes espaços de projeto. Nesses casos, uma alternativa é limitar o intervalo dos parâmetros de configuração ou quantizá-los (Gries, 2004). Por exemplo, na abordagem proposta por Givargis et al (2002), os parâmetros de configuração do espaço são agrupados de acordo com suas dependências e a busca exaustiva é realizada sobre cada grupo resultante. Em (Kulkarni et al, 2009), os autores usam técnicas para reduzir a quantidade de soluções exploradas a partir da categorização dos códigos gerados, possibilitando a exploração exaustiva do espaço dentro de um tempo considerado viável para praticamente todas as funções manipuladas.

- *Busca por amostragem aleatória*: essa estratégia é a forma mais simples de lidar com espaços grandes de projeto, na qual uma amostra do espaço é escolhida aleatoriamente. Além do processo totalmente aleatório, o qual não costuma ser muito eficiente, também consideramos nessa categoria os métodos que direcionam sua busca de forma aleatória, tais como: Monte Carlo (Bruni et al, 2001) e *simulated annealing* (Palermo et al, 2005).
- *Busca direcionada*: essa estratégia utiliza informações referentes ao domínio do problema (espaço de busca) para melhorar a convergência para as soluções ótimas globais (ou próximas do ótimo) (Gries, 2004). Esse conhecimento pode ser atualizado a cada iteração do algoritmo ou ser uma característica inerente do algoritmo de busca. Processo de decisão de Markov (Beltrame et al, 2010) e modelagem de superfície de resposta (Palermo et al, 2009) (Silvano et al, 2010) são alguns exemplos desse tipo de técnica. Além disso, é comum a utilização de métodos heurísticos como, por exemplo, *hill climbing* (Lahiri et al, 2000) e algoritmos evolutivos (Hoste e Eeckhout, 2008)(Ferrandi et al, 2008)(Ascia et al, 2011).

A Figura 3.2 ilustra o comportamento dessas estratégias de busca na cobertura de um espaço de projeto bidimensional (Gries, 2004).

Figura 3.2: Exploração em função da estratégia de busca adotada.

As abordagens exaustiva e aleatória propiciam uma busca não direcionada e não tendenciosa do espaço de projeto. Em contrapartida, a busca direcionada pode reutilizar as informações dos pontos já visitados para definir o próximo passo da busca. Outra diferença entre as estratégias refere-se a quantidade de pontos analisados em cada iteração da busca (Gries, 2004). Enquanto as buscas exaustiva e aleatória analisam uma única solução por vez, os métodos baseados no conhecimento do domínio tendem a utilizar várias configurações para encontrar uma configuração melhor.

Diferentes métodos de exploração foram utilizados em nossos experimentos a fim de avaliar suas capacidades em encontrar boas sequências de otimização, capazes de alcançar baixas latências para as funções e o processador especificados. Essas estratégias de DSE são descritas no Capítulo 5.

3.5 Métodos de Avaliação

Os métodos de avaliação utilizados no DSE devem ser capazes de comparar o desempenho das soluções testadas e fornecer as informações necessárias para verificar se os requisitos e as restrições do projeto estão sendo atendidos. A precisão e a velocidade da avaliação influenciam na eficiência da exploração. Uma boa precisão permite diferenciar duas alternativas do espaço de soluções em função de seus desempenhos, possibilitando assim identificar as melhores soluções dentre aquelas sugeridas. Já a velocidade da avaliação está associada à abstração do problema explorado. Abordagens baseadas em modelos com alto nível de abstração são mais rápidas e fáceis de implementar, mas são menos precisas. Com a evolução do projeto, algoritmos de estimação mais refinados e precisos podem ser adotados. Em contrapartida, eles são mais difíceis de construir e demandam um maior custo computacional (Sigdel, 2011).

A avaliação das soluções potenciais pode ser computada a partir de aferição (*profiling*), simulação, predição ou uma abordagem híbrida que combina algumas dessas técnicas. A aferição (Becker et al, 2003) (Genser et al, 2009) (Fursin et al, 2011) consiste em medir a métrica de desempenho desejada através da execução da aplicação diretamente na arquitetura/plataforma alvo. A forma mais comum de aferição é a adoção de ferramentas de *software* específicas que monitoram o estado do computador durante a execução do programa e a partir dos dados coletados forneçam os valores para as métricas de desempenho. Entretanto, no projeto de sistemas embarcados, é comum o uso de arquiteturas que não possuem esse tipo de ferramenta. Nesses casos, a aferição pode ser feita por instrumentação, ou seja, a utilização de equipamentos de medição externos ou componentes de *hardware* (tipicamente contadores) adicionados ao circuito do sistema para medir os valores desejados durante a execução da aplicação. A principal vantagem da instrumentação está na alta precisão das medições. Entretanto, esse tipo de medição só pode ser feito após a síntese do circuito. Portanto, essa abordagem é um método de medi-

ção de baixo nível e, devido ao tempo necessário para essa síntese, geralmente é inviável para a avaliação das soluções durante o DSE, dado que compromete a busca eficiente do espaço de soluções, ou eleva consideravelmente o tempo do DSE. Os métodos preditivos (Chabbi et al, 2011) (Meeuws et al, 2011) (Al-Khatib e Abdi, 2015) avaliam as soluções em um único passo através de modelos analíticos ou empíricos. Apesar de mais rápidos, os métodos preditivos são menos precisos devido às simplificações utilizada na modelagem do problema. Os métodos baseados em simulação (Kulkarni et al, 2009) (Jantz e Kulkarni, 2013) são particularmente adequados para investigar os efeitos dinâmicos, esporádicos e imprevisíveis no sistema (Gries, 2004). Eles necessitam de um modelo executável capaz de emular arquitetura alvo durante a execução do código. Embora apresentem um custo computacional maior, os resultados simulados são mais precisos que aqueles resultantes de modelos preditivos. O uso de métodos híbridos (Dubach, 2009), tais como análises baseadas em *trace* ou modelos preditivos com calibração inicial, buscam reduzir o tempo de avaliação através da caracterização da aplicação em uma única ou poucas simulações iniciais. Em sistemas embarcados, a avaliação por simulação é uma prática comum e, portanto, também será empregada no escopo deste trabalho.

Uma das principais premissas de um compilador é sempre gerar um modelo correto (Cardoso e Diniz, 2008). Portanto, para qualquer entrada fornecida, o código otimizado deve retornar a mesma resposta que sua versão original. Assim, uma verificação formal é fundamental em qualquer processo de compilação. No projeto de sistemas embarcados, a simulação é um dos métodos mais usados para a verificação e validação de projetos (Gajski et al, 2009)(Marwedel, 2010). Além de auxiliar na validação do modelo, a simulação também é bastante utilizada na estimativa das métricas de desempenho empregadas na avaliação da qualidade de uma solução potencial. Nesse contexto, a simulação envolve o monitoramento do comportamento não-funcional do projeto (ex: tempo de execução), o que aumenta o nível de complexidade dos simuladores (Marwedel, 2010).

A simulação consiste da execução do código compilado em um ambiente apropriado. Esse ambiente pode ser o mesmo que será utilizado na operação ou um modelo emulado em um computador convencional (PC). Durante a simulação, uma versão executável do código é estimulada e seu comportamento é monitorado. O estímulo é um conjunto de valores atribuídos aos argumentos de entrada do código simulado. A acurácia da avaliação é influenciada pelos valores dos estímulos, os quais devem representar a carga de trabalho típica do código. O simulador é responsável por propagar essas entradas, realizar as computações desencadeadas e monitorar os eventos gerados, propagando-os pelo modelo. Essa propagação dispara novas computações que, por sua vez, podem ocasionar novos eventos. Esse processo se repete até que as variáveis de saída sejam atualizadas. Caso alguma saída obtida seja diferente do valor esperado, o simulador deve indicar o erro encontrado (Gajski et al, 2009).

3.6 Trabalhos Correlatos

O problema de ordenação dos passos de otimização usados na compilação já vem sendo estudado há muitos anos (Cooper et al, 1999) (Touati e Barthou, 2006) (Quéva, 2007) (Jantz e Kulkarni, 2013). Diversas pesquisas vêm sendo feitas com o objetivo de desenvolver métodos capazes de tratar razoavelmente o problema de ordenação de fases em compiladores otimizantes (Kulkarni, 2007). A aplicação de um mesmo conjunto de otimizações em um ordem fixa, a qual é previamente ajustada pelos desenvolvedores de compiladores com base em sua experiência, ainda é uma prática comumente empregada (Kulkarni e Cavazos, 2012). Entretanto, diferentes fragmentos de código podem requerer uma ordenação específica de otimização para alcançarem o melhor desempenho para uma dada arquitetura/plataforma. Compilações para sistemas computacionais embarcados são ainda mais dependentes das transformações de código aplicadas, devido às grandes restrições de recursos, relacionados com organização e tamanho das memórias e velocidade do processador, e aos altos níveis de personalização (ex: uso de FPGAs) impostos em seus projetos arquiteturais (Cardoso et al, 2010). Nesse tipo de sistema, geralmente são necessárias sequências de otimização diferenciadas para atingir os níveis de desempenho esperados para a aplicação (Diniz e Cardoso, 2011). Apesar da evolução da área de compiladores otimizantes, a investigação por métodos que explorem de forma eficiente e efetiva o espaço de sequências de otimização ainda é um tópico relevante de pesquisa, principalmente na área de sistemas embarcados.

Diferentes abordagens de otimização de desempenho de aplicações podem ser encontradas na literatura (Hoste e Eeckhout, 2008) (Chabbi et al, 2011) (Chen et al, 2012) (Kulkarni e Cavazos, 2012) (Jantz e Kulkarni, 2013). Embora todas essas investigações anteriores foquem na obtenção de uma boa configuração dos passos de compilação, as abordagens divergem significativamente entre si. Para que seja possível elaborar um panorama comparativo dos métodos já investigados e ao mesmo tempo contextualizar a nossa abordagem dentro desse panorama, iremos definir algumas propriedades que utilizamos na sua classificação. Dividimos essas propriedades em 2 grupos:

1. **Escopo do problema:** são as propriedades que determinam como a abordagem lida com o problema de otimização de uma aplicação. Três características definem o escopo do problema no qual a abordagem foi investigada:
 - **Formulação** basicamente, existem duas formulações principais para o problema de definição do plano de otimização aplicado pelo compilador (Kulkarni, 2007): (i) seleção, na qual a ordem dos passos é fixa e a abordagem apenas decide quais passos serão efetivamente aplicados dependendo do código dado como entrada; e (ii) ordenação, na qual a abordagem seleciona quais passos serão aplicados a partir do espaço de transformações e ainda define a

melhor ordem para a aplicação dos mesmos. Além disso, alguns dos passos de otimização podem precisar da definição de valores para seus parâmetros de configuração. Por exemplo, a definição do fator de desdobramento no *loop unrolling*.

- **Tipo de compilação:** uma importante característica do problema abordado se relaciona ao tipo de compilação na qual a otimização de passos será aplicada. A compilação pode ser: (i) estática, na qual todo o programa é compilado para depois ser executado; ou (ii) dinâmica (ou JIT - *Just in Time*), na qual o programa é compilado durante sua execução. O tipo de compilação tem grande influência na abordagem de DSE empregada, uma vez que na compilação estática um tempo maior de exploração é aceitável, enquanto na compilação dinâmica, técnicas que demandam um elevado tempo de exploração se tornam inviáveis;
- **Granularidade da otimização:** algumas abordagens se preocupam em identificar uma sequência de otimização que seja suficientemente boa (representativa) para um grupo/classe de programas. Outras abordagens, por sua vez, buscam identificar sequências de otimização específicas que resultem no melhor desempenho para uma dada aplicação. Além disso, essas abordagens podem ser aplicadas tanto no nível do programa (todo o código), quanto no nível de fragmentos do código, como por exemplo as funções/métodos que compõem a aplicação.

Além dessas propriedades, outras quatro também caracterizam o escopo do problema no qual a abordagem foi aplicada: o ambiente de compilação (compilador), o número de passos de otimização, além dos *benchmarks* e a plataforma alvo utilizados durante a avaliação da abordagem.

2. **Técnica adotada:** englobam as propriedades que definem os métodos empregados na solução do problema. Cinco características foram identificadas para descrever as técnicas utilizadas:

- **Modo de Compilação:** a identificação das sequências de otimização geralmente são baseadas em compilação iterativa (Cooper et al, 2006) (Chen et al, 2012) (Jantz e Kulkarni, 2013) ou métodos baseados em predição (Agakov et al, 2006) (Fursin et al, 2011) (Kulkarni e Cavazos, 2012). A primeira consiste de diversas compilações do código fonte, cada qual usando uma sequência diferente de passos de otimização, e a escolha daquela que atingir o melhor desempenho para a métrica avaliada. Embora a busca iterativa possa melhorar efetivamente o desempenho das aplicações, ela costuma demandar um grande

número de compilações durante a exploração. Métodos baseados em predição normalmente compreendem o uso de modelos treinados para selecionar sequências de otimização para maximizar o desempenho com base nas características do código. Esse tipo de abordagem reduz significativamente o tempo de exploração, enquanto que os métodos baseados em compilação iterativa tipicamente obtêm códigos otimizados com os melhores desempenhos (Ashouri et al, 2014). Em ambas abordagens, técnicas de aprendizagem de máquina têm sido comumente empregadas na busca das melhores sequências de otimização para aplicações específicas.

- **Etapas de caracterização:** grande parte dos modelos de aprendizado de máquina realizam predições com base em características extraídas do código a ser compilado ou das sequências de otimização potenciais. Assim, na descrição de uma abordagem preditiva é importante determinar que tipo de informação deve ser extraída do código a ser compilado.
- **Forma de avaliação:** qualquer que seja a abordagem empregada no DSE, é importante avaliar a qualidade da solução proposta. Essa avaliação pode ser realizada através da execução efetiva do código otimizado diretamente na plataforma alvo, através da sua simulação, ou ainda, por meio de modelos preditivos. As duas últimas abordagens são geralmente aplicadas quando a execução direta não é possível ou seu custo computacional é muito alto.
- **Método de redução do espaço:** devido ao imenso espaço de busca envolvido no processo de DSE, algumas das abordagens realizam um pré-processamento com o objetivo de reduzir o espaço de busca, antes de executar o DSE.
- **Método de busca:** a principal etapa do DSE refere-se à busca propriamente dita, ou seja, a investigação da sequência de passos de compilação que retorna um bom desempenho para uma dada aplicação;

A Tabela 3.1 apresenta uma descrição das abordagens investigadas na literatura, a partir das propriedades que caracterizam o escopo do problema: formulação, tipo de compilação, granularidade, compilador, número de passos, *benchmarks* e plataforma alvo. Cada linha da tabela refere-se a uma abordagem anterior diferente, identificada pela referência do trabalho na primeira coluna. A última linha apresenta as propriedades da abordagem desenvolvida em nossa pesquisa.

Tabela 3.1: Características do problema abordado pelos trabalhos correlatos.

Artigo	Formulação do problema	Tipo de compilação	Granularidade	Compilador	Número de Passos	Benchmarks	Plataforma alvo
Agakov <i>et al.</i> , 2006	Ordenação	Estática	Função	SUIF (<i>source to source</i>) + <i>backend</i> (<i>code composer</i> ou GCC)	14 (considerando 4 fatores de <i>loop unrolling</i>)	12 <i>kernels</i> do UTDSP	TI C6713 e AMD Au1500
Ashouri <i>et al.</i> , 2014	Seleção	Estática	Programa	GCC-ARM	7	24 benchmarks do cBench	ARM
Cavazos e O'Boyle, 2006	Seleção	Dinâmica	Função	Jikes RVM	20	SPECjvm98 e DaCapo	Jikes RVM
Cavazos <i>et al.</i> , 2007	Seleção	Estática	Programa	PathScale EKOPath	121	SPEC (int e fp), Mibench e Polyhedron	x86 (AMD Athlon)
Chabbi <i>et al.</i> , 2011	Ordenação	Estática	Função	Não informado	13	fmin (código fortran)	Não informado
Chen <i>et al.</i> , 2012	Seleção	Estática	Programa	GCC e ICC	132	32 variações do MiBench	Intel Xeon E3110
Cooper <i>et al.</i> , 1999	Ordenação	Estática	Programa	C/FORTRAN <i>frontend</i> e SPARC <i>backend</i>	10	8 programas FORTRAN e 6 programas C	Sparc Ultra-10
Fursin <i>et al.</i> , 2011	Seleção	Estática	Programa	GCC (<i>Milepost</i>)	88	Cbench e Berkeley DB	x86 (AMD e Intel) e ARC
Hoste e Eeckhout, 2008	Seleção	Estática	Classes de programas (níveis de otimização)	GCC	60 e 3 especiais (<i>stripped</i>)	SPEC CPU2000	Não informado
Jantz e Kulkarni, 2013	Ordenação	Estática	Função	VPO	15	MiBench (2 por categoria)	ARM
Kulkarni e Cavazos, 2012	Ordenação (iterativa)	Dinâmica	Função	Jikes RVM	14	SPECjvm98, SPECjvm2008 e DaCapo	x86 (2 proc. dual core AMD Opteron)
Kulkarni, 2007	Ordenação	Estática	Função	VPO	15	6 benchmarks do Mibench	ARM
Kulkarni <i>et al.</i> , 2010	Ordenação	Estática	Função	VPO	15 (considerando 3 fatores de <i>loop unrolling</i>)	12 benchmarks do Mibench	ARM
Lima <i>et al.</i> , 2013	Ordenação	Estática	Programa	LLVM	51	Polybench	x86 (intel Core i7)
Purini e Jain, 2013	Ordenação	Estática	Classes de programas (níveis de otimização)	LLVM	62	Microkernels, MiBench e PolyBench	Intel Xeon W35550
Sanchez <i>et al.</i> , 2011	Seleção	Dinâmica	Função	Testarossa (IBM J9 JVM)	Não Informado	SPECjvm98 e DaCapo	x86 (16 proc. Quad-core AMD Opteron)
Sher <i>et al.</i> , 2014	Ordenação (iterativa)	Estática	Função e programa	LLVM	53 (programa) e 34 (função)	ccbench e bzip	x86 (intel Core i7)
Nossa abordagem	Ordenação	Estática	Função	ReflectC (CoSy e LARA)	49	TI DSP e TI IMG	Xilinx MicroBlaze

A tabela 3.2 apresenta as mesmas abordagens, a partir das propriedades que caracterizam a técnica empregada: modo de compilação, caracterização, avaliação, redução do espaço, método de busca. Novamente, cada linha da tabela refere-se a uma abordagem da literatura, sendo que a última linha refere-se à nova abordagem desenvolvida. A seguir, detalharemos alguns dos trabalhos caracterizados nas tabelas, destacando-se os trabalhos mais relacionados à nossa pesquisa.

Pesquisas recentes buscam propor novas técnicas heurísticas baseadas em vetor de características para customizar rapidamente sequências para funções individuais durante a compilação JIT (*Just In Time*). Em (Cavazos e O'Boyle, 2006), um modelo preditivo baseado em regressão logística usa as características estáticas extraídas da função para selecionar que otimizações devem ser aplicadas em cada método pelo *Jikes RVM Java*, usando uma ordem fixa de passos. Máquinas de vetores suporte (em inglês, *Support Vector Machines* ou SVMs) (Faceli et al, 2011) foram empregadas por Sanchez et al (2011) para derivar sequências de otimização específicas para funções visando a máquina virtual JAVA (JVM - *Java Virtual Machine*) J9 da IBM. Tanto Cavazos e O'Boyle (2006), quanto Sanchez et al (2011), usam modelos específicos para cada nível de otimização. Essas abordagens revelam que o uso de modelos analíticos e funções de custo com base em certas características do código permitem bons preditores e podem realmente contribuir para a identificação de sequências de otimizações em compilação. O trabalho descrito por Cavazos e O'Boyle (2006) foi estendido por Kulkarni e Cavazos (2012) a fim de lidar com a ordenação dos passos de compilação para cada método especificamente. Os autores tratam o problema da ordenação como um processo de Markov, no qual a caracterização do estado corrente do código é usada como entrada de uma rede neural artificial para serem selecionadas as próximas otimizações a serem aplicadas durante uma compilação dinâmica.

Diferente do escopo de nossa pesquisa, os métodos anteriores interagem com o compilador a fim de montar a sequência dinamicamente. Nesse processo, o próximo passo de otimização a ser aplicado é escolhido a partir do estado atual do código (abordagem gulosa). Entretanto, nossa abordagem é focada em compilação estática.

Tabela 3.2: Características da abordagem adotada em cada um dos trabalhos correlatos.

Artigo	Forma de avaliação	Características extraídas	Dados de saída	Modo de Compilação	Método de redução do espaço	Método de Busca
Agakov <i>et al.</i> , 2006	Execução	Características no nível de <i>loop</i> extraídas por PCA	Sequência personalizada de otimização	Híbrida	Método de seleção da função mais próxima	Busca aleatória e AG baseados em modelos probabilísticos
Ashouri <i>et al.</i> , 2014	Execução	Características dinâmicas (MICA)	Conjunto personalizado de otimização	Híbrida	Nenhum	Redes Bayesianas
Cavazos e O'Boyle, 2006	Execução	Características estáticas	Conjunto de probabilidades de aplicação de cada otimização	Modelo preditivo	Nenhum	Regressão Logística
Cavazos <i>et al.</i> , 2007	Execução	Características dinâmicas (60 contadores de desempenho)	Conjunto personalizado de otimização	Híbrida	Regressão Logística	Amostragem a partir da distribuição de probabilidade
Chabbi <i>et al.</i> , 2011	Predição	Nenhum	Sequência personalizada de otimização	Compilação iterativa	Modelo baseado em grafo direcionado para mapear o relacionamento entre pares de otimizações	3 abordagens para direcionamento da busca
Chen <i>et al.</i> , 2012	Execução	Nenhum	Sequência de otimização (não personalizada)	Compilação iterativa	Nenhum	Amostragem aleatória
Cooper <i>et al.</i> , 1999	Execução	Nenhum	Sequência personalizada de otimização	Compilação iterativa	Nenhum	Algoritmo Genético (AG)
Fursin <i>et al.</i> , 2011	Execução (<i>profiling</i> no nível de função)	Características estáticas (56)	Conjunto personalizado de otimização	Modelo preditivo	Nenhum	Modelo probabilístico e árvore de decisão
Hoste e Eeckhout, 2008	Execução	Nenhum	Conjunto personalizado de otimização	Compilação iterativa	Nenhum	AG multi-objetivo (variação do SPEA2 com múltiplas populações)
Jantz e Kulkarni, 2013	Simulação (execução para validar resultado)	Nenhum	Sequência personalizada de otimização	Compilação iterativa	Redução baseada na independência entre os passos	Busca exaustiva em múltiplos estágios e algoritmo genético
Kulkarni e Cavazos, 2012	Execução	Caracterização do estado atual do programa (26)	Conjunto de probabilidades de aplicação de cada otimização	Modelo preditivo	Nenhum	Processo Markoviano baseado em redes neurais artificiais
Kulkarni, 2007	Simulação	Nenhum	Sequência personalizada de otimização	Compilação iterativa	Remoção da redundância no espaço de sequências	Algoritmo Genético
Kulkarni <i>et al.</i> , 2010	Simulação	Nenhum	Sequência personalizada de otimização	Compilação iterativa	Remoção da redundância no espaço de sequências	busca exaustiva e AG
Lima <i>et al.</i> , 2013	Execução	Características dinâmicas (contadores de desempenho)	Sequência personalizada de otimização	Híbrida	Amostragem aleatória e filtragem pelos requisitos	Seleção de sequências baseada em casos (similaridade)
Purini e Jain, 2013	Execução	Caracterização das sequências baseada em uma matriz de desempenho	Sequência de otimização (não personalizada para a função)	Compilação iterativa	Seleção de sequências relevantes	Busca exaustiva sobre o conjunto reduzido
Sanchez <i>et al.</i> , 2011	Execução	Características estáticas (71)	Conjunto personalizado de otimização	Modelo preditivo	Nenhum	Máquinas de vetores de suporte (SVM)
Sher <i>et al.</i> , 2014	Execução	Características estáticas a nível de função (44) ou a nível de programa (48)	Conjunto de probabilidades de aplicação de cada otimização	Híbrida	Nenhum	Processo Markoviano baseado em redes neurais artificiais
Nossa abordagem	Simulação	DNA do código	Sequência personalizada de otimização	Abordagem híbrida	Seleção de passos baseado em agrupamento	Diversas estratégias de DSE

Como exemplo desse tipo de abordagem, um sistema chamado COLE é apresentado por Hoste e Eeckhout (2008), que usa um algoritmo evolutivo multi-objetivo baseado no SPEA2 (Zitzler et al, 2001) para encontrar automaticamente as sequências de otimização Pareto-ótimas para o compilador GCC. O trabalho descrito por Chen et al (2012) demonstrou experimentalmente que existe pelo menos uma seleção de otimizações no nível de programa que atinge 83% ou mais do melhor *speedup* possível dentre 1000 conjuntos de dados diferentes para cada um dos 32 programas considerados. Os autores sugerem que a otimização dos programas através de diferentes conjuntos de dados pode ser mais simples que as previsões anteriores. Sher et al (2014) estende a abordagem apresentada por Kulkarni e Cavazos (2012) para determinar sequências de otimização para o compilador LLVM. Ainda no campo das abordagens estáticas, um modelo baseado em redes Bayesianas foi usado por Ashouri et al (2014) para prover a distribuição de probabilidades para os passos de compilação a partir das características independentes de máquina dinamicamente extraídas da aplicação. Essas probabilidades guiam a busca iterativa pelo auto-ajuste de *flags* de otimização do compilador GCC, almejando a aceleração da execução da aplicação.

Tanto as abordagens estáticas quanto as dinâmicas (baseadas em compilação JIT) citadas anteriormente revelaram que o uso de modelos e funções de custo baseados em determinadas características de código pode prover boas métricas de predição e ajudar na busca das melhores sequências de otimizações. Por outro lado, a nossa abordagem centra-se na seleção de passos potenciais de otimização a fim de conseguirmos restringir/reduzir o espaço de projeto para a execução posterior do método de busca da DSE. Por isso, o nosso trabalho é ortogonal a estas abordagens uma vez que a poda do espaço de busca pode ser vista como uma tarefa separada do processo de exploração.

A redução pode ser utilizada como um importante passo inicial para a aplicação das estratégias de DSE. Várias técnicas foram investigadas para reduzir o espaço de soluções candidatas. Por exemplo, a identificação de casos limítrofes no espaço de busca pode ser realizada por sondagem do espaço de projeto ou por métodos analíticos de pior caso. Em (Almagor et al, 2004), um subespaço de busca é formado com sequências de otimização de comprimento 10, considerando-se apenas 5 otimizações distintas. Esse subespaço foi exaustivamente enumerado para alguns pequenos programas. Cada uma dessas enumerações tipicamente requer vários meses de processamento, mesmo para pequenos programas. Os autores descobriram que tais espaços de busca não eram nem planos nem convexos, bem como notaram a dificuldade em prever as melhores otimizações na maioria dos casos.

A análise estatística do efeito de opções do compilador é usada por Haneda et al (2005) para podar o espaço de busca e encontrar uma sequência de otimizações única para um conjunto de programas, cujo desempenho seja melhor que aquele obtido com as configurações padrão utilizadas no GCC.

Agakov et al (2006) utilizam técnicas de aprendizado de máquina para limitar o espaço de busca, aumentando a velocidade das otimizações iterativas. A redução do espaço é realizada a partir do agrupamento baseado na métrica de similaridade do vizinho (função) mais próximo. Numa segunda etapa, o DSE é realizado através de uma busca (aleatória ou baseada em algoritmo genético) direcionada por modelos probabilísticos, construídos a partir das boas sequências conhecidas para a função mais próxima. Essa metodologia é capaz de indicar, para um determinado programa, as regiões do espaço de soluções em que a busca deve ser focada. Além disso, como em nossa abordagem, ela é independente do espaço de solução, do algoritmo de busca e do compilador/otimizador utilizados.

Cavazos et al (2007) apresentam uma extensão de seu trabalho anterior (Cavazos e O'Boyle, 2006) para tratar a seleção de otimizações no compilador *PathScale EKOPath*. Nessa nova abordagem, os autores adotam contadores de desempenho para mapear o comportamento dinâmico dos programas, identificando aspectos importantes do seu desempenho, tais como: utilização das unidades de ponto flutuante, falha de cache (*cache misses*), estatísticas de predição de desvios (*branch prediction*), entre outros. Esses contadores são utilizados para encontrar similaridade entre os programas e identificar um conjunto de passos de otimização que podem melhorar a eficiência do novo programa. Segundo os autores, uma pequena amostra das otimizações que podem ser geradas a partir desse conjunto reduzido de passos é suficiente para melhorar a eficiência do código compilado.

Kulkarni (2007) representa o espaço de sequências na forma de um grafo acíclico direcionado (DAG - *Directed Acyclic Graph*), possibilitando assim a aplicação de técnicas de poda agressivas para explorar as redundâncias no espaço das funções (códigos otimizados) através da detecção de otimizações latentes e de pontos idênticos/equivalentes. Segundo os autores, a poda usada praticamente viabiliza a avaliação exaustiva de todas as sequências de otimização para a maioria das funções. Então, vários algoritmos de aprendizado de máquina são usados para encontrar sequências eficazes, e os resultados obtidos (desempenhos e custos de cada abordagem) são comparados entre si e com o melhor desempenho encontrado. Essa análise permitiu observar que o uso de técnicas de busca, como o algoritmo genético, permitem obter ganhos muito próximos ao melhor desempenho. No processo de exploração, métodos de estimação baseados na categorização das funções de acordo com seus fluxos de controle de bloco básico foram usados para agilizar a avaliação das sequências. Esse trabalho foi estendido por Kulkarni et al (2010), onde buscas baseadas em AG e em nível de função foram comparadas em relação à sua capacidade em encontrar as melhores sequências de otimização para o compilador VPO (*Very Portable Optimizer*) (Benitez e Davidson, 1988). Jantz e Kulkarni (2013) utilizam a abordagem de compilação implementada por Kulkarni et al (2010), empregando um algoritmo DSE de busca exaustiva, adicionando características independentes dos passos de otimização em uma técnica de redução, realizando buscas em estágios múltiplos em cima

de pequenos subconjuntos disjuntos de passos de otimização.

Um ambiente de compilação baseado no GCC (Milepost GCC) é usado por Fursin et al (2011) para extrair automaticamente as características do código e aprender as melhores otimizações para vários programas e arquiteturas. O Milepost utiliza técnicas de aprendizado de máquina para correlacionar o novo programa com aquele que for o mais próximo dentre os vistos anteriormente para aplicar uma combinação de otimizações personalizada e potencialmente mais eficaz. Embora esse último trabalho seja bem próximo à nossa própria abordagem, nosso método não precisa de nenhum treinamento prévio do modelo preditivo para incluir novos conhecimentos ou passos de otimização (abordagem não supervisionada (Faceli et al, 2011)), sendo baseado apenas no agrupamento a partir do conjunto de referência e na aplicação de esquemas simples de busca no processo de DSE. Assim, a adaptabilidade do método (aprendizado de novo conhecimento ou alteração daquele já existente) consiste apenas na modificação do conjunto de referência adotado, sem a necessidade de reconstrução ou novo treinamento (*off-line*) dos modelos empregados que, por sua vez, costumam demandar uma grande massa de dados e tempo computacional. Além disso, em nossa abordagem não é necessário definir as características que devem ser extraídas explicitamente do código fonte, embora as mesmas possam ser utilizadas como dados de entrada durante a fase de agrupamento para melhorar a precisão do algoritmo. A ideia é que os padrões de código extraídos implicitamente pelo agrupamento auxiliam na seleção apropriada de passos de otimização, os quais podem ser utilizados nas estratégias de DSE para conduzir/sugerir sequências de otimização eficientes.

Técnicas de sub-amostragem (*down-sampling*) são usadas em (Purini e Jain, 2013) para reduzir o espaço de busca para um conjunto pequeno de boas sequências de otimização (em torno de 10) que seja representativo para todas as classes de programa (uma espécie de engenharia reversa sobre o problema de classificação). Durante a compilação, todas as sequências desse subconjunto são aplicadas no novo programa e a melhor delas é então escolhida. Esse trabalho difere do nosso, pois nossa abordagem é focada na busca de uma sequência personalizada para uma função.

A estratégia COSPpp descrita por de Lima et al (2013) é uma abordagem de seleção baseado em casos que seleciona automaticamente um conjunto de passos de otimização para um novo programa a fim de melhorar o desempenho e o consumo de potência obtidos pelos níveis de otimização do compilador. COSPpp adota um algoritmo de busca baseado em similaridade para encontrar boas sequências de otimização a partir de um espaço reduzido que contém informações prévias sobre um grupo de programas. Para cada programa existente no espaço, são registrados os contadores de desempenho coletados a partir do seu código original (não otimizado), bem como uma amostra de sequências de otimização geradas aleatoriamente. O processo de busca consiste em selecionar um pequeno subconjunto de sequências de otimização com base na classificação (*rank*) gerada a partir da similaridade dinâmica entre programas. Todas sequências do subconjunto são

avaliadas (busca exaustiva) e aquela que melhor atender aos requisitos de desempenho e consumo de potência é adotada. Assim como nossa abordagem, esse trabalho também utiliza agrupamento para selecionar um subconjunto de sequências de otimização. Entretanto, enquanto os autores tratam essas sequências de forma isolada, nosso método as utiliza para gerar um único conjunto de passos de otimização, o qual é empregado pelo algoritmo de busca.

Além das abordagens caracterizadas nas tabelas 3.1 e 3.2, outros trabalhos na literatura também endereçam o problema da otimização do código compilado (Kisuki et al, 1999), (Cooper et al, 2006), (Willcock et al, 2009), (Meeuws et al, 2011), (Dragomir, 2011), (Boudec e Thiran, 2001), (Richter et al, 2002), (Pan e Eigenmann, 2008). Entretanto, essas últimas abordagens citadas divergem significativamente da abordagem aqui proposta.

Capítulo 4

Seleção Baseada em Agrupamento

O código fonte tem características que tornam difícil o uso direto de métodos de agrupamento típicos, principalmente em relação à sua dependência das propriedades extraídas de código para a mineração (Sanches et al, 2011). Para lidar com esse problema, o código fonte é convertido para uma representação simbólica, denominada DNA de um programa (Sanches e Cardoso, 2008), devido a sua semelhança com as sequências de DNA usadas na genética (Sanches e Cardoso, 2010). Após a conversão, o DNA é submetido a um processo de mineração de dados para a identificação das similaridades entre os códigos.

Nesta tese, adotamos uma variação do método de mineração de dados apresentado por Sanches et al (2011). Esse método é baseado em técnicas de agrupamento e permite encontrar padrões nos fragmentos de códigos, independentemente do seu tamanho ou linguagem de programação. Por exemplo, em nossos experimentos foi usada uma granularidade no nível de função e códigos escritos na linguagem C. Além disso, ele aceita diferentes dados como entrada, tais como: representações intermediárias, métricas e/ou outras informações coletadas do código. Porém, a acurácia do resultado obtido está diretamente ligada a qualidade dessa entrada.

Nossa abordagem combina três diferentes algoritmos para gerar agrupamentos com o novo código (função alvo) e as funções usadas como referência. Inicialmente, uma matriz é gerada a partir do cálculo das distâncias entre cada par de código de DNA. Nesse processo foi empregada a métrica da Distância de Compressão Normalizada (NCD - *Normalized Compression Distance*) (Cilibrasi e Vitanyi, 2005). Um algoritmo de reconstrução de árvores filogenéticas, chamado Junção de Vizinhos (do inglês *Neighbor Joining* ou NJ) (Felsenstein, 2003), utiliza essa matriz de distâncias para contruir a topologia da árvore. Então, um algoritmo de agrupamento baseado em ambiguidade identifica grupos de funções altamente correlacionadas, extraíndo potenciais agrupamentos escondidos na topologia da árvore filogenética. Por fim, os diferentes passos de otimização usados nas sequências adotadas pelas funções de referência agrupadas são incluídos como espaço de busca ou configuração inicial no DSE. A exploração pode então ser executada através de qualquer dos métodos de busca implementados no ambiente do compilador, tais como

o algoritmo genético (Goldberg, 1989) ou o algoritmo de inserção (Huang et al, 2013).

A Figura 4.1 ilustra o processo de seleção dos passos de otimização baseado no agrupamento e na representação por DNA. Esse processo é responsável por gerar o conjunto de otimizações usados pelas estratégias de DSE, conforme será ilustrado na Figura 5.7.

Figura 4.1: Seleção dos passos de otimização do compilador baseado em agrupamento.

O pseudocódigo desse processo de seleção é apresentado no Algoritmo 4.2. Suas entradas são: um repositório com a representação do código (DNA_{Ref}) e a sequência de passos de compilação (Seq_{Ref}) para cada função presente no conjunto de referência; o código fonte da nova função a ser compilada (New_{code}); e o limiar de ambiguidade (T_{Amb}) usado pelo algoritmo de agrupamento. A saída do algoritmo é um subconjunto dos passos de otimização existentes no espaço de busca ($S_{Reduzido}$).

O processo de seleção descrito no algoritmo é baseado em quatro etapas principais: codificação do DNA a partir do código fonte, cálculo da matriz de distâncias através da métrica NCD, construção da árvore genética usando o algoritmo NJ, e agrupamento baseado em ambiguidade. Cada uma dessas etapas são descritas a seguir.

4.1 Codificação do DNA

O efeito de um passo de otimização durante a compilação é altamente dependente do código sendo compilado (Kulkarni, 2007)(Jantz, 2010). Portanto, a identificação de padrões de código pode expor similaridades entre as funções e pode ajudar a determinar as potenciais sequências de otimizações. Entretanto, a mineração de dados para identificar tais padrões sobre o código fonte pode ser uma tarefa muito complexa, dado que pode ter de lidar com muitos arquivos, cada qual com inúmeras linhas de código e diferentes estruturas de código (Sanches e Cardoso, 2010). Muitas pesquisas em mineração de padrões (Wheeler e Kececioğlu, 2007)(Roy et al, 2009) baseiam-se em níveis elevados de similaridade

Entrada: Conjunto de referência (DNA_{Ref} e Seq_{Ref}); o código fonte da nova função (New_{code}); e o limiar de ambiguidade (T_{Amb})

Saída: Espaço reduzido de passos de compilação ($S_{Reduzido}$)

```

1 Início
   // Codificação do DNA
2    $tokens \leftarrow AST(New_{code});$ 
3    $DNA_{New} \leftarrow \{\};$ 
4   Para todo  $x \in tokens$  faça
5      $DNA_{New}.incluir(convert(x));$ 
6   Fim-para
   // Cálculo da Matriz de distâncias ( $Dist$ )
7    $DNA \leftarrow DNA_{Ref} \cup \{DNA_{New}\};$ 
8    $sz \leftarrow tamanho(DNA);$ 
9    $Dist \leftarrow zeros(sz, sz);$ 
10  Para  $x = 1$  até  $sz - 1$  faça
11    Para  $y = x + 1$  até  $sz$  faça
12       $d_{xy} \leftarrow ncd(DNA(x), DNA(y));$ 
13       $d_{yx} \leftarrow ncd(DNA(y), DNA(x));$ 
14       $Dist(x, y) \leftarrow Dist(y, x) \leftarrow (d_{xy} + d_{yx}) \div 2;$ 
15    Fim-para
16  Fim-para
   // Construção da árvore filogenética ( $Tree$ )
17   $Tree \leftarrow arvoreEstrela(Dist);$ 
18  Enquanto  $tamanho(Dist) > 3$  faça
19     $\{A, B\} \leftarrow obtmParVizinhos(Dist);$ 
20     $R \leftarrow juntaVizinhos(Tree, A, B);$ 
21     $calculaNovosRamos(Tree, R, A, B);$ 
22     $atualiza(Dist);$ 
23  Fim-enquanto
   // Agrupamento baseado em ambiguidade
24   $Cluster \leftarrow \{\};$ 
25  Para todo  $x \in DNA_{Ref}$  faça
26     $qt \leftarrow caminhoNodes(Tree, DNA_{New}, x);$ 
27    Se  $tamanho(qt) < T_{Amb}$  então
28       $Cluster.incluir(\{x, qt\});$ 
29    Fim-se
30  Fim-para
31   $Cluster.ordena(qt);$ 
32   $S_{Reduzido} \leftarrow \{passos(Cluster.node(1))\};$ 
33  Para  $x = 2$  até  $tamanho(Cluster)$  faça
34     $Seq \leftarrow passos(Cluster.node(x));$ 
35    Para todo  $y \in Seq$  faça
36      Se  $y \notin S_{Reduzido}$  então
37         $S_{Reduzido}.incluir(y);$ 
38      Fim-se
39    Fim-para
40  Fim-para
41  Retorna  $S_{Reduzido};$ 
42 Fim

```

Figura 4.2: Algoritmo de seleção de passos de compilação baseado em agrupamento.

que prejudicam a identificação de padrões entre códigos com diferenças sutis, mas que possivelmente compartilham a mesma estratégia de otimização.

Nós empregamos a codificação em DNA devido a sua capacidade em destacar as principais estruturas do código de acordo com as regras de transformação especificadas, permitindo identificar padrões aproximados a partir do código fonte. O DNA é uma representação simbólica apresentada em (Sanches e Cardoso, 2008)(Sanches e Cardoso, 2010)(Sanches et al, 2011), na qual os elementos dos códigos (ex: operadores e loops) são codificados em uma *string* de símbolos, de modo similar a genética.

A conversão é baseada em regras de transformação e gera representações intermediárias simplificadas do código original (com alguma perda de informação). A escolha do conjunto de regras de transformação pode influenciar no grau de precisão com o qual a representação do DNA reflete a sequência de *tokens* que identifica cada fragmento de código. Inicialmente, usamos o mesmo conjunto de regras de transformação empregado em (Sanches et al, 2011), o qual enfatiza as similaridades das operações no corpo dos *loops*. Quando os *loops* estão hierarquicamente aninhados no código, eles são convertidos para uma única linha de código do DNA equivalente, sendo que o *loop* mais externo é colocado na frente do *loop* mais interno. A conversão de códigos fonte com *loops* sequenciais gera representações de DNA com uma linha por *loop*. Entretanto, durante as investigações foram avaliadas algumas variações do conjunto original, as quais incluem novas regras de transformação (ex: regras direcionadas à tipos de dados). Esses novos conjuntos de regras, possivelmente mais complexos, foram testados a fim de verificar o impacto de novas regras na acurácia do processo de DSE. Os resultados desses experimentos são apresentados no Capítulo 6.

Os passos da codificação em DNA são apresentados na Figura 4.2, entre as *linhas* 2 e 6. Inicialmente é realizada a análise (*parsing*) do código fonte e a geração da árvore sintática abstrata (AST - *Abstract Syntax Tree*). Cada *token* da AST é então traduzido para a representação em DNA através do conjunto de regras de transformação adotado.

A Figura 4.3 ilustra cinco exemplos hipotéticos de fragmentos de código com estruturas de repetição (*loops*), suas respectivas representações em DNA, e o conjunto original de regras de transformação (Sanches et al, 2011).

4.2 Distância de Compressão Normalizada

A Distância de Compressão Normalizada (NCD), é uma métrica proposta em (Cilibrasi e Vitanyi, 2005) para o cálculo de distância baseada na compressão de arquivos, que vem sendo bastante utilizada na área de mineração de dados (Cilibrasi e Vitanyi, 2005)(Wehner, 2007)(Klenk et al, 2009)(Ito et al, 2010)(Sanches et al, 2011). Ela computa a diferença (dissimilaridade) entre objetos, sendo suas propriedades inspiradas na teoria da complexidade de Kolmogorov (Li e Vitanyi, 1997). Entretanto, diferentemente do método de Kolmogorov,

Código C		Representação em DNA
1	int i, o = A; for (i=0; i<G; i++) { o+= o; } }	HIEZICIQIPI
2	int j, p = B; for (j=0; j<H; j++) { p*= p; } }	HIEZICIQIAEI
3	int k, l, q = C; for (k=0; k<M; k++) { for (l=0; l<N; l++) { q*= q; } } }	HIEZICQHIEZICIQIAEI
4	int m, n, s = E; for (m=0; m<K; m++) { for (n=0; n<L; n++) { s+= s; } } }	HIEZICQHIEZICIQIPI
5	int u, v, x, y = O; for (u=0; u<O; u++) { for (v=0; v<P; v++) { for (x=0; x<Q; x++) { y*= y; } } } }	HIEZICQHIEZICQHIEZICIQIAEI

Regras de Transformação:			
"for"	→ "H"	"++"	→ "Q"
Identificador	→ "I"	"*", "x"	→ "A"
"+="	→ "P"	Op. relacional	→ "C"
"="	→ "E"	Leitura([])	→ "R"
"0"	→ "Z"	Escrita([])	→ "W"

Figura 4.3: Exemplos de codificação do DNA a partir do código C.

a NCD tem um custo computacional razoável. A ideia básica é que dois objetos são considerados próximos se a informação do primeiro pode ser significativamente compactada a partir da informação do segundo, e vice-versa. Portanto, a distância $NCD(x; y)$ para cada par de funções (x e y) é a melhoria decorrente da compressão de y usando x como referência e da compressão de y sem qualquer informação de referência, expressa como a razão entre o comprimento (*bit a bit*) das duas versões compactadas. Essa métrica pode ser calculada da seguinte forma (Cilibiasi e Vitanyi, 2005):

$$NCD(x; y) = \frac{C(xy) - \min(C(x); C(y))}{\max(C(x); C(y))} \quad (4.1)$$

sendo $C(\star)$ o comprimento da versão compactada do arquivo $\star$, e xy o arquivo resultante da junção dos conteúdos de x e y .

A métrica NCD não requer nenhuma informação sobre as características (propriedades) dos dados analisados para encontrar a relação entre eles. Essa independência de conhecimento prévio permite que o algoritmo baseado em NCD seja aplicado sobre vários tipos de informação do programa a nível do código, incluindo (mas não limitado a) os códigos fonte, código *Assembly* e representações intermediárias (como o DNA). Isto também possibilita que o algoritmo extraia informações do relacionamento entre os códigos em diferentes níveis de abstração (Sanches e Cardoso, 2010).

O algoritmo calcula a métrica NCD a partir do código DNA e gera a matriz de distâncias

Tabela 4.1: Matriz de distâncias NCD a partir da representação DNA.

DNA	1	2	3	4	5
1	0	0.29	0.36	0.21	0.40
2	0.29	0	0.20	0.34	0.31
3	0.36	0.20	0	0.28	0.25
4	0.21	0.34	0.28	0	0.35
5	0.40	0.31	0.25	0.35	0

$N \times N$, onde N é o número de funções manipuladas (referências e alvo).

Como apresentado na Figura 4.2, os elementos da matriz são zerados (*linha 9*) e, para cada par de funções x e y , a métrica NCD é calculada através da média aritmética entre $NCD(x; y)$ e $NCD(y; x)$ (*linhas 10 a 16*). A média aritmética é usada para garantir a simetria da matriz resultante.

A Tabela 4.1 apresenta a matriz de distâncias obtida a partir dos códigos de DNA descritos na Figura 4.3. Através da matriz é possível observar, por exemplo, que os códigos 2 e 3 (usando suas representações DNA) são os mais similares de acordo com o NCD, enquanto que os códigos 1 e 5 são os mais diferentes.

Vários compressores podem ser usados no cálculo da NCD. Inicialmente, utilizamos o compressor *bzlib* para calcular as matrizes de distâncias dos experimentos. Essa seleção foi baseada em experimentos preliminares sobre as funções usadas em (Sanches et al, 2011), nos quais o uso do compressor *bzib* resultou nos melhores agrupamentos. Esses experimentos foram repetidos posteriormente sobre as funções dos *benchmarks* para avaliar o impacto dos outros compressores disponíveis no pacote Complearn (Cilibrasi et al, 2008), tais como: *zlib*, *bzlib*, *ppmdx* e *lzmax*, na exploração dos benchmarks testados.

4.3 Algoritmo de reconstrução de árvores filogenéticas

A reconstrução de árvores filogenéticas parte do princípio que todos os indivíduos sob observação descendem de um mesmo ancestral comum e, portanto, é possível representar suas relações evolutivas através de uma estrutura do tipo árvore (da Silva, 2007). A topologia da árvore define o padrão de ancestralidade de modo que cada espécie ocupe uma folha da árvore e os nós internos representam os ancestrais das subárvores ligadas a ele. O comprimento dos ramos (ramificação) indicam o número de mudanças evolutivas que ocorreram desde um ancestral comum. Ambos são hipóteses descritivas de eventos não-observáveis e condicionais, ou seja, pode existir mais de uma configuração (topologia e ramificação) adequada a partir de um mesmo conjunto de dados (da Silva, 2007). Devido à essa característica não determinística, a inferência da melhor configuração (topologia e ramificação) para uma árvore filogenética é uma tarefa complexa e que exige consome muito tempo e esforço. Portanto, algoritmos computacionais são usualmente empregados

na construção de árvores capazes de representar adequadamente as diferenças (dissimilaridades) entre os objetos sob comparação (da Silva, 2007).

A árvore filogenética pode ser com ou sem raiz, sendo que os procedimentos de reconstrução geralmente específicos para cada caso (da Silva, 2007). A existência de raiz implica na definição ou conhecimento prévio da ordem em que ocorreu a diferenciação evolutiva (Felsenstein, 2003). Como isso não se aplica ao problema de identificação de padrões entre os códigos das funções, adotamos algoritmos que operam com árvores sem raiz, mais especificamente o algoritmo *Neighbor Joining* (NJ) (Felsenstein, 2003). Esse algoritmo é bastante utilizado, devido a sua simplicidade e boa eficiência computacional.

O NJ é um algoritmo baseado no princípio da mínima evolução, o qual visa minimizar o comprimento global da árvore (soma do comprimento de todos os ramos da árvore). A topologia da árvore filogenética é construída iterativamente através de consecutivos agrupamentos dos pares de nós vizinhos. A ideia básica é que a topologia resultante permita determinar recursivamente agrupamentos compostos por outros grupos. O algoritmo também determina o comprimento dos ramos da árvore. A Figura 4.4 ilustra a construção iterativa de uma árvore filogenética.

Conforme apresentado no algoritmo da Figura 4.2, o NJ transforma uma matriz de distâncias (entrada) em uma estrutura árvore sem raiz que descreve os relacionamentos entre os objetos sob comparação. Esse processo inicia a partir de uma árvore-estrela (*linha 17*), na qual todos as funções (nós folha) são conectados por um nó interno comum. Em cada iteração, um par de vizinhos (dois nós folha) é selecionado, considerando seu potencial para a redução do comprimento geral da árvore, e ligados através de um novo nó intermediário que representa o seu ancestral comum (*linhas 19 a 21*). Então, em cada junção de vizinhos, o número de nós folha é decrementado e a matriz de distâncias é recalculada de modo a considerar esse novo nó (*linha 22*). Essas junções são executadas sucessivamente enquanto a árvore-estrela possuir mais de 3 nós, ou seja, até se obter uma matriz de distâncias 3×3 .

Embora a estratégia gulosa de redução iterativa do comprimento da árvore não garanta atingir o valor mínimo (ótimo global), o algoritmo NJ é competitivo em relação aos métodos mais recentes de reconstrução de árvores filogenéticas (da Silva, 2007)(Sanches et al, 2011). Além disso, como o NJ não depende de qualquer tipo de conhecimento *a priori* sobre o domínio do problema, ele é interessante para a identificação da hierarquia de similaridades entre as funções.

4.4 Agrupamento Baseado em Ambiguidade

Dada a árvore de relacionamentos resultante do algoritmo NJ, é importante verificar a existência de subárvores com um alto grau de independência. Portanto, um algoritmo de agrupamento é empregado para extrair os agrupamentos potenciais que estão escondidos

Figura 4.4: Exemplo da reconstrução de árvore filogenética através do Algoritmo *Neighbor Joining*.

na topologia da árvore.

Ao contrário da abordagem apresentada por Sanches et al (2011), na qual foi utilizado o algoritmo *Fast Newman* (FN) (Newman, 2010), nós adotamos um método de agrupamento inspirado na medida de ambiguidade em estruturas do tipo árvore. O FN é um algoritmo para análise de redes complexas que foca na descoberta de grupos em redes de larga escala (ex: comunidades em redes sociais). A abordagem baseada em ambiguidade, por outro lado, responde melhor na análise de pequenas redes. Inicialmente, nossa expectativa é lidar com um número pequeno de funções de referência, tornando a abordagem baseada em ambiguidade mais apropriada. Entretanto, se for necessária a adoção de conjuntos de referência maiores, é possível migrar para o algoritmo FN.

A ambiguidade de uma árvore pode ser determinada a partir da ambiguidade entre dois nós folha da seguinte forma:

Os nós folha i e j de uma árvore T são ambíguos, se existem dois ou mais nós intermediários (ancestrais) no único caminho entre i e j de acordo com a topologia da árvore T .

Em outras palavras, dois objetos (i e j) não são tão semelhantes, se exigem mais de um nó interno para contrabalançar as suas diferenças em relação a outros objetos, os quais compõem uma terceira força que afeta o equilíbrio entre i e j .

A ambiguidade da árvore T (ou ambiguidade acumulada) é a soma das ambigüidades de todos os pares de folhas de T . Esta definição pode resultar em uma avaliação imprecisa da confiabilidade de um agrupamento. Qualquer árvore com três nós tem ambiguidade igual a 1 (um). Além disso, qualquer árvore com mais de uma folha e com um número ímpar de nós folha tem, pelo menos, ambiguidade igual a 1. Para árvores maiores (com pelo menos algumas centenas de folhas), esta ambiguidade adicional pode ser irrelevante quando se compara a confiabilidade das subárvores do agrupamento. Entretanto, isto deve ser evitado quando se avalia árvores pequenas, como aquelas utilizadas em nosso ambiente.

Uma forma de contornar esse problema é a adoção do conceito de ambiguidade forte.

Basicamente, dois nós folha são considerados fortemente ambíguos se existirem três ou mais nós intermediários entre eles. Nossa abordagem de agrupamento emprega medidas de ambiguidade forte para definir quais funções de referência pertencem ao grupo da nova função de entrada (alvo). Neste contexto, a função de entrada é agrupada a todas funções de referência acessíveis por até dois nós intermediários.

A Figura 4.5 mostra o agrupamento realizado sobre a árvore filogenética construída a partir dos cinco exemplos de DNA da Figura 4.3, considerando a amostra 5 como função de entrada.

Figura 4.5: Agrupamento baseado no conceito de ambiguidade forte.

Neste caso, a amostra 5 (função alvo) é agrupada com a amostra 3 por um nó interno (*A*), com a amostra 2 através de um caminho formado por dois nós intermediários (*A* e *B*), e com as amostras 1 e 4 através de um caminho formado por três nós intermediários (*A*, *B* e *C*). Considerando o conceito de ambiguidade forte, o terceiro nó interno (nó *C*) é a fronteira do grupo. Portanto, a amostra 5 foi agrupada com todos os nós folha separados por um caminho com até dois nós internos (nós *A* e *B*), ou seja, amostras 3 e 2.

O número de nós intermediários permitidos (limiar de ambiguidade) pode ser modificado, possibilitando que a força do relacionamento necessária para estabelecer um agrupamento possa ser definida de acordo com o domínio investigado. Já nos primeiros experimentos sobre os *benchmarks*, ajustamos o limiar de ambiguidade para 4 (quatro), de modo a evitar grupos com apenas uma função de referência. Dessa forma, os grupos passaram a ser formados pelas funções de referências cujo o caminho em relação à função alvo seja formado por até três nós intermediários, ou seja, que estejam no escopo das subárvores ilustradas na Figura 4.6.

No algoritmo da Figura 4.2, a etapa de agrupamento começa com um grupo vazio (*linha 24*). Então, o comprimento do caminho entre a função alvo e cada função de referência é calculado, ou seja, contabiliza a quantidade de nós interno entre elas (*linha 26*). Se esse comprimento for menor que o limiar de ambiguidade (T_{Amb}), o nó (função de referência)

Figura 4.6: Subárvores formadas pelo agrupamento com até três nós intermediários.

é incluído no grupo (*linhas 27 a 29*). Caso contrário, o nó é descartado. Ao final, as funções de referência agrupadas são ordenadas de acordo com o comprimento do seu caminho até a função alvo (*linha 31*), e suas otimizações são incluídas no conjunto de saída (espaço de otimizações) de acordo com a ordem das funções e dos passos de otimização na sequência (*linhas 32 a 40*).

Capítulo 5

Estratégias de DSE

A capacidade do método em explorar soluções viáveis e encontrar a melhor é influenciada pela metodologia adotada e as ferramentas empregadas no ambiente de compilação (Sigdel, 2011). Os métodos devem auxiliar na avaliação das diferentes sequências de otimização, permitindo identificar qual é o melhor valor de latência que pode ser alcançado pelo código compilado no processador almejado.

Em nossos experimentos, diferentes estratégias de exploração foram implementadas com o objetivo de avaliar sua capacidade em encontrar sequências de otimização que propiciassem alcançar baixas latências para as funções compiladas. Essas estratégias diferem em relação à técnica e ao espaço de busca empregados na exploração da melhor sequência de otimização para uma dada função.

Este capítulo descreve as estratégias de exploração do espaço de projeto desenvolvidas e avaliadas nesta pesquisa, descrevendo como cada uma trata o conjunto de passos de otimização gerado pelo nosso método de seleção baseada em agrupamento. Além disso, apresentamos a infraestrutura de DSE usada para desenvolver essas estratégias e aplicá-las durante o processo de compilação.

5.1 Algoritmos Genéticos

Algoritmos evolutivos são métodos heurísticos de natureza estocástica que combinam busca aleatória com a sobrevivência do mais apto durante a criação de novos indivíduos a partir de uma população inicial de soluções. Nessa abordagem, o conhecimento do domínio deve ser aplicado para evitar a geração de soluções inválidas (Gries, 2004). Diversos trabalhos (Cooper et al, 1999)(Kulkarni, 2007)(Hoste e Eeckhout, 2008)(Ferrandi et al, 2008)(Ascia et al, 2011) utilizam algum tipo de algoritmo evolutivo durante a exploração de grandes espaços de projeto.

Nesta tese, uma estratégia de DSE baseada em algoritmos genéticos (AG) foram usados para explorar o espaço de busca e encontrar a sequência de otimizações ótima para

cada função a ser compilada. Algoritmos genéticos são métodos computacionais de busca e otimização caracterizados pela inspiração na biologia evolutiva, por meio da simulação de mecanismos como a seleção natural e herança genética (Goldberg, 1989). Eles simulam a teoria da seleção natural de Darwin de forma limitada sendo, portanto, uma simplificação do processo natural. A ideia básica desse algoritmo é que indivíduos mais aptos tendem a sobreviver e se reproduzir, e que os filhos sejam ao menos tão bem adaptados em seu ambiente quanto seus pais.

O funcionamento típico de um AG é ilustrado pelo fluxograma da Figura 5.1 (Mitchell, 1996).

Figura 5.1: Funcionamento típico de um Algoritmo Genético.

Inicialmente, uma população inicial é criada, a qual consiste em um conjunto de indivíduos gerados aleatoriamente. Um indivíduo, ou cromossomo, representa uma possível solução para o problema investigado. Essa representação pode ser estática (estrutura com tamanho fixo e pré-definido) ou dinâmica (estrutura com tamanho variável). A forma

de representação adotada também influencia nos valores que podem ser atribuídos a cada posição do cromossomo (gene). Por exemplo, enquanto algumas abordagens adotam uma representação binária visando facilitar a aplicação das operações genéticas, outras preferem assumir valores reais dos parâmetros de configuração do espaço de soluções. Após a geração, os indivíduos da população inicial são avaliados através de uma função de aptidão (*fitness*), a qual está relacionada com a métrica objetivo a ser otimizada.

O processo de evolução é executado iterativamente a partir da população inicial, sendo cada passo evolutivo chamado de geração. Esse processo é repetido até que um critério de parada seja atingido. O ideal seria que o AG parasse quando um ponto ótimo fosse encontrado. Entretanto, geralmente não é possível determinar se o ótimo global foi encontrado sem realizar uma busca exaustiva, o que não costuma ser inviável. Nesses casos, critérios de parada do AG devem ser definidos, tais como: número máximo de gerações ou tempo limite de processamento. Outro critério plausível é parar o algoritmo usando a ideia de estagnação, ou seja, quando não se observa melhoria da população depois de várias gerações consecutivas (Goldberg, 1989).

A cada iteração do algoritmo, o operador de seleção escolhe pares de soluções (pais), sobre os quais são aplicados as operações genéticas (ex: *crossover* e mutação) a fim de produzir novos indivíduos (filhos). O objetivo dessas operações é a geração de novas soluções para regiões mais promissoras do espaço de busca. O operador de *crossover* mistura pedaços de um par de pais para gerar um par de filhos. O operador de mutação provoca ocasionalmente pequenas alterações nos filhos, visando prover maior diversidade na população. Enquanto o primeiro busca combinar características de duas boas soluções para gerar outras melhores (busca guiada), a mutação provoca perturbações nos genes das soluções geradas, possibilitando a inclusão de novidades no cromossomo (fator aleatório). Esses operadores genéticos são condicionados a uma probabilidade, a qual determinam sua efetiva aplicação.

Sempre que um novo indivíduo é gerado, ele precisa ser avaliado, ou seja, seu valor de aptidão deve ser calculado. Após a geração da população de filhos, o AG precisa determinar quais soluções, entre pais e filhos, irão sobreviver para a próxima geração. Isto é feito pelo operador de reinserção, o qual também pode ter um forte impacto na convergência do AG para soluções ótimas. De um modo geral, dentre as soluções existentes no ambiente em um dado momento, apenas aquelas mais aptas conseguem persistir e passar seu material genético adiante. Assim, a população tende a melhorar ao longo das gerações.

As principais etapas do processo iterativo do AG são brevemente descritas a seguir.

5.1.1 Seleção de indivíduos

O objetivo da seleção é escolher pais que representem boas soluções, de acordo com as suas aptidões, e criar filhos ainda mais aptos à sobrevivência no ambiente. Existem vários

algoritmos de seleção, mas os mais utilizados na literatura são a roleta e o torneio.

No método da roleta, cada indivíduo da população recebe uma quantidade de números proporcional ao seu valor de aptidão. A seleção de um indivíduo consiste no sorteio aleatório de um número. Assim, indivíduos com maior aptidão tem maior chance de serem selecionados. Esse processo é realizado repetidamente até que o número de pais necessários para o *crossover* seja obtido.

No método do torneio, a seleção é feita a partir de grupos de indivíduos escolhidos sucessivamente da população corrente. O indivíduo mais adaptado dentro de cada grupo é selecionado como resultado do torneio. O modo como os grupos são formados determina o tipo do torneio: simples ou estocástico. No torneio simples, a seleção dos indivíduos do grupo é feita de forma aleatória, sem considerar a aptidão de cada um. No torneio estocástico, a seleção do grupo é feita através da roleta. O tamanho dos grupos (*tour*) é um parâmetro de configuração do AG, sendo usualmente adotados grupos de 2 ou 3 indivíduos.

5.1.2 *Crossover*

O *crossover* é um processo que estabelece a troca de material genético entre pais para a formação dos filhos. Essa operação é controlada por um parâmetro, conhecido como taxa de *crossover*, que indica a porcentagem de indivíduos da população inicial que será utilizada como pais na geração de novos indivíduos. A formação dos filhos depende de dois aspectos a serem definidos durante o cruzamento: a quantidade e a localização dos pontos de corte.

A forma de representação do cromossomo influencia diretamente na escolha do operador de *crossover*, uma vez que sua aplicação pode resultar em uma combinação de valores inválida, ou seja, que não atende às especificações do problema. Por exemplo, considere que o cromossomo representa uma sequência de números inteiros sem repetição, o uso do *crossover* apresentado na Figura 5.2 pode ocasionar genes com números repetidos, invalidando os indivíduos gerados. Nesses casos, deve-se adotar um operador que seja capaz de lidar com repetições, como o *crossover* cíclico ou o PMX (Goldberg, 1989).

Mutação

A mutação modifica um ou mais genes de alguns indivíduos gerados no *crossover*. A escolha de qual indivíduo sofrerá mutação, bem como dos genes a serem modificados é totalmente aleatória. A mutação é controlada por um parâmetro conhecido por taxa de mutação, que indica a probabilidade de um indivíduo ou de um gene sofrer mutação.

A forma como um gene sofre mutação também depende da estrutura de representação do cromossomo. Por exemplo, em uma representação binária, a mutação consiste no complemento dos *bits* selecionados.

Reinserção da população

O processo de reinserção ocorre no final de cada geração e consiste em selecionar os indivíduos que serão mantidos na população, ou seja, que sobreviverão para a próxima geração. Esse processo é necessário para reiniciar o ciclo da evolução, uma vez que a criação de uma nova geração de soluções candidatas (filhos) aumenta a quantidade de indivíduos da população, extrapolando o tamanho máximo previamente estabelecido. O tamanho da população também é um parâmetro de configuração do AG que influencia na sua convergência e no tempo de execução. Quanto maior for a população, mais pontos do espaço de busca são avaliados em cada geração, aumentando a cobertura da exploração. Em contrapartida, como o tempo gasto na avaliação é geralmente o gargalo do processo de busca, o aumento no número de indivíduos avaliados por geração onera o tempo de execução do algoritmo. Encontrar o melhor compromisso (*trade-off*) entre o tamanho da população e tempo de execução é um dos desafios durante a escolha dos parâmetros do AG.

Os principais métodos da reinserção são:

- **Reinserção pura:** substitui-se toda a população antiga pelos filhos.
- **Reinserção uniforme:** selecionam-se os melhores indivíduos da população total (pais e filhos) a partir de um dos métodos de seleção (roleta ou torneio).
- **Reinserção baseada na aptidão:** a população total é classificada (*ranking*) e selecionam-se os melhores indivíduos.
- **Elitismo:** manutenção de parte da população com melhor avaliação (elite) para a próxima geração, sendo o restante da população obtida por reinserção pura ou uniforme.

5.1.3 Configuração do AG

A eficiência do AG está diretamente associada à escolha dos seus parâmetros de configuração. Essa configuração é usualmente definida de forma empírica, através da avaliação de diferentes configurações e seleção daquela que resulta no melhor compromisso entre convergência e tempo de execução. As configurações dos AGs usados em nossa abordagem são descritas a seguir.

Crítérios de parada

o AG é executado por no máximo 100 gerações ou até que a métrica objetivo (latência do código) não seja melhorada por 15 gerações subsequentes. Durante os experimentos, o número máximo de gerações nunca foi atingido e, mesmo assim, o AG foi capaz de

encontrar sequências de otimização que resultassem no melhor desempenho obtido em cada função. Isso indica que o valor adotado é mais que suficiente para lidar com o problema da forma como foi modelado. A quantidade de gerações sem melhoria foi escolhida considerando o compromisso entre tempo de execução e acurácia (melhoria nas latências das funções compiladas). Um número menor de gerações provoca a parada prematura do algoritmo para muitas das funções exploradas, enquanto que seu aumento eleva demasiadamente o tempo de exploração, ao mesmo tempo que melhorou o resultado obtido em poucas execuções para algumas funções.

População do AG

Durante as gerações do AG é mantida uma população de tamanho fixo e pré-definido. Cada indivíduo dessa população (cromossomo) é um ponto no espaço de busca (sequência de otimização do compilador) representado como um vetor de tamanho variável que corresponde ao comprimento da sequência. Indivíduos de tamanho fixo também foram testados. Entretanto, apesar de facilitar o uso dos operadores genéticos, a necessidade de incluir um novo passo de compilação para indicar que nenhuma otimização deve ser aplicada em determinadas posições da sequência aumenta consideravelmente o tempo da avaliação e torna mais complexa a comparação entre dois indivíduos para identificar sequências já exploradas (uma mesma sequência pode ser obtida a partir de indivíduos diferentes). Cada posição do cromossomo (gene) deve conter um passo de otimização, indicando a ordem pela qual ele será usado pelo compilador.

Durante a geração da população inicial, sequências de diferentes comprimentos são geradas respeitando uma distribuição uniforme, de modo a garantir quantidades similares para todos os possíveis tamanhos (sequências de 1 a 16 otimizações). O tamanho máximo das sequências foi definido em experimentos preliminares com as mesmas funções usadas por Sanches et al (2011). Enquanto sequências menores reduzem bastante a média das latências obtidas para as funções, o uso de sequências maiores resultaram em melhorias insignificantes que não justificam o aumento no tempo de exploração (maior sequência implica em mais tempo na avaliação).

Todos os indivíduos da população são avaliados e classificados (ranqueados) de acordo com as suas aptidões. Neste trabalho, a latência (em número de ciclos do relógio) associada ao código gerado, após a aplicação da sequência de otimização representada pelo indivíduo, foi utilizada como métrica de avaliação.

Etapas da evolução

Em cada geração, os pares de indivíduos usados no *crossover* são selecionados pelo método de torneio simples baseado em grupos de 3 indivíduos ($T_{our} = 3$).

Cada par selecionado tem 60% de probabilidade (taxa de *crossover*) de ser submetido a uma operação de *crossover*, conforme ilustrado na Figura 5.2. Esse operador utiliza um ponto de corte para cada pai, o qual é selecionado aleatoriamente. Os pares de filhos são gerados a partir da concatenação cruzada das partes dos cromossomos dos pais. O filho 1 é gerado pela concatenação da primeira parte do cromossomo do pai 1 com a segunda parte do cromossomo do pai 2. O filho 2 é gerado pela concatenação da primeira parte do cromossomo do pai 2 com a segunda parte do cromossomo do pai 1.

Figura 5.2: Operador de *crossover* aplicado pelo AG.

Uma taxa de mutação de 40% foi aplicada sobre a população de filhos (novos indivíduos gerados pelas operações de *crossover*). A Figura 5.3 apresenta os quatro tipos de operadores de mutação que foram desenvolvidos, os quais são descritos a seguir:

- **Inclusão de uma nova otimização:** esse operador inclui um novo passo de otimização em uma posição da sequência escolhida aleatoriamente. O passo a ser inserido também é escolhido de forma aleatória dentre aqueles existentes no conjunto de otimizações considerado na exploração. Essa operação aumenta o tamanho da sequência de otimização original, exceto quando ela já tem o tamanho máximo permitido, provocando então sua substituição.
- **Substituição de uma otimização:** nessa operação, uma posição da sequência é selecionada de forma aleatória, a qual recebe o passo de otimização escolhido aleatoriamente do conjunto explorado.
- **Remoção de uma otimização:** nessa operação, uma posição do cromossomo é selecionada, como no operador de substituição, e seu conteúdo é removido da sequência.
- **Mudança na ordem de aplicação:** esse operador troca 2 otimizações da sequência, sendo que suas localizações (posição) são escolhidas aleatoriamente como nos demais operadores de mutação.

Todos os operadores de mutação têm a mesma probabilidade de ocorrência e a escolha entre eles é totalmente aleatória. Para cada indivíduo submetido à mutação, apenas um operador é aplicado. Operações de mutação também são aplicadas sobre indivíduos repetidos (já avaliados), visando gerar uma nova solução ainda não explorada. Nesses casos, três operadores de mutação são aplicados sobre o indivíduo.

Figura 5.3: Operadores de mutação aplicados pelo AG.

Em nossa abordagem foi empregado o método de reinserção baseada na aptidão. Essa estratégia de reinserção mantém as melhores sequências de otimização (pais e filhos) na população ao final de cada geração, garantindo sua propagação para as gerações seguintes.

5.2 Amostragem Aleatória

A amostragem aleatória é a forma mais simples de exploração encontrada na literatura (Gries, 2004). Ele consiste em selecionar um subconjunto de amostras do espaço de soluções. Dada a complexidade do problema da compilação iterativa, algumas pesquisas apontam que as sequências de otimização geradas de forma aleatória são tão boas quanto aquelas obtidas a partir de métodos heurísticos, como algoritmos genéticos ou *simulated annealing* (Agakov et al, 2006)(Chen et al, 2012)(Ashouri et al, 2014). A forma como essa seleção é realizada pode influenciar na acurácia da abordagem. Uma seleção totalmente aleatória costuma apresentar uma acurácia inferior àquela que adota algum tipo de distribuição. Em nossa estratégia de DSE, a amostragem segue exatamente os mesmos passos usados na geração da população inicial pelo AG. Isto é, escolhe aleatoriamente os passos de otimização que ocuparão cada posição das sequências, cujo os tamanhos são definidos seguindo uma distribuição uniforme.

Esse tipo de abordagem normalmente precisa de uma grande quantidade de amostras para obter soluções razoáveis. Para garantir uma boa amostragem, adotamos um conjunto $10\times$ maior que o tamanho da população usado nas abordagens baseadas em algoritmo genéticos. Assim, para a exploração do conjunto completo de otimizações, a abordagem baseada em amostragem aleatória utiliza 3000 amostras. Na exploração do conjunto reduzido gerado pelo método baseado em agrupamento, são geradas 300 amostras. Em nossos experimentos exploratórios, observamos que os tamanhos adotados foram suficientes para atingir boas sequências de otimização (próximas àquelas encontradas pelo

AG) para ambos os conjuntos (completo e reduzido), uma vez que amostragens maiores só aumentam o tempo de exploração sem melhorar o desempenho da abordagem. No caso do conjunto reduzido, o número de otimizações disponíveis no conjunto é relativamente pequeno (13 em média). Além disso, esse conjunto é formado a partir da "filtragem" do conjunto completo por nossa abordagem de seleção, sendo esperado que ele contemple somente otimizações potenciais para o código sob compilação. Assim, 300 amostras, mesmo que geradas aleatoriamente, é suficiente para alcançar bons resultados a partir desse conjunto.

5.3 Algoritmo de Inserção

O algoritmo de *inserção* é um processo iterativo guloso apresentado por (Huang et al, 2013), o qual inclui incrementalmente novos passos de otimização à sequência atual. Seu pseudocódigo é apresentado na Figura 5.4.

Inicialmente, o código original (sem otimização) é simulado e a latência obtida é usada como referência para o processo de otimização. Então, o algoritmo percorre sequencialmente o conjunto de otimizações, em uma ordem pré-definida, construindo a solução (sequência) através da inserção de um passo de otimização por vez e da verificação do desempenho resultante a partir do código otimizado. Nesse processo, cada otimização é avaliada em todas as possíveis posições da sequência, visando obter a melhor configuração. O desempenho dessa nova sequência (com o passo de otimização) é então comparado com aquele obtido pela sequência atual (sem o passo de otimização). Se houver uma melhoria, a otimização é mantida na solução em construção. Caso contrário, ela é descartada e o próximo passo de otimização é testado.

O algoritmo de *inserção* pode ser utilizado na exploração do conjunto completo de passos de otimização ou do subconjunto gerado pelo nosso método de seleção (conjunto reduzido). No conjunto completo, a ordem dos passos de otimização é definida arbitrariamente. No conjunto reduzido, a ordem das otimizações é definida de acordo com a proximidade entre as funções na árvore filogenética. Isto é, nós adotamos a mesma ordem empregada na sequência de compilação da função de referência mais próxima, incluindo as otimizações distintas (novas) encontradas nas sequências das outras funções do grupo. O critério adotado para definir a proximidade entre 2 funções está relacionada com a quantidade de nós intermediários existentes no caminho entre elas (conceito de ambiguidade, Seção 4.2). Uma vez que as otimizações são processadas na mesma ordem que aparecem no conjunto, o desempenho da abordagem baseada em inserção pode ser influenciado pelo arranjo adotado. Para minimizar esse problema, é necessário percorrer o conjunto de otimizações um número maior de vezes (N), sendo N um parâmetro de entrada do algoritmo. A adoção de um número pequeno de iterações pode aumentar a dependência entre a ordem do conjunto e os resultados, limitando a cobertura do espaço de busca. Por outro lado,

Entrada: Número de iterações (N), espaço de busca (S) e função alvo (F)
Saída: Melhor sequência de otimização para o compilador ($bestSeq$)

```

1 Início
2    $currSeq \leftarrow \{\}$ ;
3    $currFit \leftarrow avalia(F, currSeq)$ ;
4    $refFit \leftarrow currFit$ ;
5   Para  $i = 1$  até  $N$  faça
6      $bestSeq \leftarrow currSeq$ ;
7      $bestFit \leftarrow currFit$ ;
8     Para  $e = 1$  até  $tamanho(S)$  faça
9       Para  $pos = 1$  até  $tamanho(currSeq) + 1$  faça
10         $newSeq \leftarrow inserePasso(S(e), currSeq, pos)$ ;
11         $newFit \leftarrow avalia(F, newSeq)$ ;
12        Se  $newFit < bestFit$  então
13           $bestSeq \leftarrow newSeq$ ;
14           $bestFit \leftarrow newFit$ ;
15        Fim-se
16      Fim-para
17      Se  $bestFit > currFit$  então
18         $currSeq \leftarrow bestSeq$ ;
19         $currFit \leftarrow bestFit$ ;
20      Fim-se
21    Fim-para
22    Se  $currFit < refFit$  então
23       $refFit \leftarrow currFit$ ;
24    Fim-se
25    else
26       $break$ 
27    end
28  Fim-para
29  Retorna  $bestSeq$ ;
30 Fim

```

Figura 5.4: Algoritmo de *Inserção*.

um número grande de iteração aumenta o tempo de execução do DSE. Neste trabalho, adotamos 3 iterações para ambas as abordagens (conjunto completo e reduzido), uma vez que valores maiores não propiciam melhores resultados em nossos experimentos. Além disso, o algoritmo também pode parar antecipadamente quando não houver melhoria entre essas iterações.

5.4 Algoritmo de Limpeza

O algoritmo de *limpeza* também é uma abordagem iterativa gulosa, a qual remove incrementalmente os passos de compilação da sequência de otimização atual. Ele é a abordagem mais simples adotada, sendo criada originalmente para limpar as sequências resultantes de outras estratégias de DSE. Entretanto, sua utilização pode ser interessante para os projetos que apresentam critérios mais relaxados de desempenho e altas restrições de *time-to-market*.

A Figura 5.5 descreve o pseudocódigo do algoritmo de *limpeza*.

Entrada: espaço de busca (S) e função alvo (F)
Saída: Melhor sequência de otimização para o compilador ($bestSeq$)

```

1 Início
2    $bestSeq \leftarrow S$ ;
3    $bestFit \leftarrow avalia(F, bestSeq)$ ;
4    $pos \leftarrow 1$ ;
5   Enquanto  $pos \leq tamanho(bestSeq)$  faça
6      $newSeq \leftarrow removePasso(bestSeq, pos)$ ;
7      $newFit \leftarrow avalia(F, newSeq)$ ;
8     Se  $newFit \leq bestFit$  então
9        $bestSeq \leftarrow newSeq$ ;
10       $bestFit \leftarrow newFit$ ;
11     Fim-se
12     Senão
13        $pos \leftarrow pos + 1$ ;
14     Fim-se
15   Fim-enquanto
16   Retorna  $bestSeq$ ;
17 Fim

```

Figura 5.5: Algoritmo de *Limpeza*.

Ao contrário das estratégias anteriores, o algoritmo de *limpeza* inicia a partir de uma sequência de compilação formada por todos os passos de otimização existentes no conjunto. A cada iteração, uma otimização é removida da sequência atual, visando verificar seu efeito no desempenho do código otimizado. Essa remoção é feita sequencialmente, desde a primeira até a última posição da sequência (N -ésimo passo de otimização). Se

houver um aumento na latência, o passo de otimização é considerado importante para o desempenho e, portanto, deve ser mantido em sua posição original. Caso contrário, assume-se que o passo de otimização não influencia no desempenho alcançado, podendo ser definitivamente removido da sequência. Assim, a execução do algoritmo envolve $N + 1$ simulações do código explorado (função alvo), sendo N o comprimento do conjunto de otimizações.

Dado que o algoritmo de *limpeza* não permite alterar a ordem das otimizações na solução, seu desempenho é fortemente influenciado pela distribuição (arranjo) dos passos de otimização na sequência original. Em nossos experimentos, essa abordagem só foi utilizada para explorar o conjunto reduzido de otimizações, no qual a ordem dos passos de compilação é definido de acordo com a proximidade das funções na árvore filogenética.

5.5 Abordagens Híbridas

Durante os experimentos, nós observamos que a utilização conjunta dos algoritmos de inserção e limpeza na estratégia de DSE melhora a efetividade da exploração, aproximando seu desempenho àquele encontrado pela estratégia de DSE baseada em algoritmo genético sobre o conjunto completo de otimizações (referência). Essa abordagem foi inspirada no fato das funções comparadas serem similares e não idênticas, possuindo, portanto, particularidades entre os códigos. Essas particularidades podem possibilitar o uso de otimizações específicas que não são encontradas em nenhuma outra função do grupo. Portanto, aplicamos uma única iteração do algoritmo de *inserção* baseado no conjunto completo de otimizações com o objetivo de lidar com otimizações que não estão presentes no conjunto reduzido. Além da maior adaptabilidade, essa abordagem também permite a reordenação das otimizações, contornando o problema identificado no algoritmo de *limpeza*. A Figura 5.6 mostra um exemplo do funcionamento das duas abordagens híbridas adotadas neste trabalho, as quais são descritas a seguir:

- **Limpeza & Inserção (CI - *Clean-Insertion*)**: essa abordagem recebe como entrada o subconjunto de otimizações gerado pelo nosso método de seleção baseado em agrupamento. Como ilustrado na Figura 5.6a, a exploração inicia com a aplicação do algoritmo de *limpeza* sobre esse conjunto reduzido (sequência inicial) a fim de remover todas as otimizações desnecessárias (que não melhoram o desempenho do código), reduzindo assim o tamanho da sequência. A sequência gerada é então usada como sequência inicial para o algoritmo de *inserção*. Uma vez que o processo de inserção parte de um ponto promissor não vazio do espaço de busca, a sequência ótima pode ser alcançada através da inclusão de poucos passos do conjunto completo de otimizações. Na prática, o uso de todas as otimizações pelo algoritmo de *inserção* permite a exploração de regiões vizinhas ao subespaço original. Final-

Figura 5.6: Exemplo de funcionamento das abordagens híbridas.

mente, a inclusão de uma nova otimização pode tornar algum outro desnecessário. Portanto, o algoritmo de *limpeza* deve ser novamente executado. Dado que esse algoritmo é simples e rápido, sua repetição não afeta significativamente o tempo de exploração desta abordagem.

- **Inserção & Limpeza (IC - *Insertion-Clean*):** essa abordagem também utiliza o subconjunto reduzido como entrada (sequência inicial). Entretanto, a exploração inicia com a inclusão de novas otimizações na sequência inicial antes da aplicação do algoritmo de *limpeza*, como mostrado na Figura 5.6b. A ideia básica é que passos de otimização redundantes podem ser combinados com novas otimizações melhorando a latência do código compilado. A abordagem IC também pode ser considerada uma forma especial do algoritmo de *inserção* sobre o conjunto completo, na qual é definida uma sequência inicial promissora com base no conjunto reduzido e reduzida a quantidade de iterações para uma única passagem sobre o conjunto completo de otimizações. Em contrapartida, o algoritmo de *inserção* lida com sequências maiores, resultando em mais posições de teste para cada nova otimização. Portanto, o tempo de execução dessa abordagem é bem maior que o da abordagem CI e o desempenho é similar.

5.6 Infraestrutura do DSE

Todas as nossas estratégias de exploração foram implementadas e aplicadas através de um ambiente de compilação integrado, o qual foi desenvolvido no contexto do Projeto REFLECT (Cardoso et al, 2013). Este ambiente inclui o ReflectC, um compilador controlado por aspectos/estratégias escritos na linguagem LARA. Ele é baseado no CoSy (*Compiler System*) (Alt et al, 1994)(ACE, 2014), um compilador industrial que suporta uma grande quantidade de otimizações.

A Figura 5.7 mostra o ambiente de compilação baseado em LARA usado neste trabalho.

Figura 5.7: Fluxo de compilação baseado em LARA.

5.6.1 Linguagem LARA

LARA (Cardoso et al, 2012) é uma linguagem orientada a aspecto de domínio específico que efetivamente implementa estratégias de DSE capazes de controlar e guiar completamente todas as ferramentas integradas (*toolchains*) no ambiente de compilação, tais como: compiladores e simuladores, assim como avaliar as alternativas de otimização com base nas respostas (*feedbacks*) relatadas durante a exploração.

As estratégias implementadas em LARA definem as otimizações específicas do compilador que melhor atendam ao mapeamento de uma aplicação para o processador embarcado desejado (ex: Xilinx MicroBlaze). Um aspecto de alto nível implementa o algoritmo de DSE a partir de um mecanismo de repetição mais externo do LARA (representado por linhas tracejadas na Figura 5.7), enquanto outros aspectos são responsáveis por guiar o compilador ReflectC, e por gerar e avaliar os códigos resultantes. A modularidade da infraestrutura do DSE requer apenas o aspecto que encapsula a execução das ferramentas do ambiente de compilação, permitindo sua modificação quando se utiliza um compilador

diferente. Esta visão integrada e unificada também permite programar e aplicar facilmente diferentes estratégias de DSE.

5.6.2 Ferramentas de compilação

As ferramentas integradas nesse ambiente de compilação (*toolchains*) são baseadas no compilador CoSy e foram construídas em torno da linguagem LARA. O uso do LARA possibilita que os desenvolvedores abstraíam a interação entre as ferramentas empregadas enquanto efetivamente guiam o processo de DSE na exploração das alternativas de otimização para o processador desejado. Isso permite lidar com uma grande quantidade de transformações e processadores alvo, mapeando as escolhas de uma forma automática (Cardoso et al, 2012). Por exemplo, um *weaver* integrado no compilador ReflectC possibilita a escolha de diferentes geradores de código, tais como: um gerador para o processador Xilinx MicroBlaze (usado neste trabalho) e outro para VHDL (ambos baseados no CoSy).

O compilador baseado no CoSy propicia o pleno controle da ordenação dos passos de compilação, bem como possibilita o ajuste desses passos através de um *weaver* escritos em LARA. Basicamente, esse *weaver* é responsável pela execução das especificações em LARA que indicam o passo de compilação a ser aplicado. A Tabela 5.1 apresenta os 50 passos de compilação suportados no ReflectC, os quais realizam análise e/ou transformações de código de baixo (dependentes da arquitetura) e alto nível. Parte desses passos foram sugeridos pelo fabricante do compilador e o restante foi selecionado pela equipe de desenvolvimento em função de sua aparente importância para o escopo do projeto REFLECT. Uma das vantagens da nossa abordagem é não precisar de qualquer informação prévia sobre as interações entre os passos de compilação. Assim, durante o desenvolvimento das estratégias de DSE não foram considerados os tipos dos passos, nem a fase na qual eles devem ser aplicados. Maiores detalhes sobre os passos de compilação do CoSy (também referenciados como *engines*) podem ser encontrados em (ACE, 2014).

Tabela 5.1: Passos de compilação do CoSy suportados pelo ambiente de compilação.

Tipo	Nome	Descrição
Análise	alias	Define campo 'Alias' para objetos
	funceval	Avalia as chamadas a funções funcionais
	loopbcount	Adiciona atributo de estimação de uso a blocos com base no aninhamento dos laços
	markconvert	Configura estrutura "mirConvert" quando o argumento assume valores válidos
	setpurity	Calcula efeitos colaterais de execução de procedimento
	setrefobj	Verifica se as variáveis locais de ponteiro referem-se a um objeto específico
	algebraic	Aplicação de diversas simplificações algébricas
	blockmerge	Combinação de blocos básicos
	cache	Armazena dados da memória em variáveis temporárias locais
	chainflow	Otimizador salto-para-salto (<i>jump-to-jump</i>)
	ckfstrength	Substitui as chamadas de subrotinas conhecidas do compilador por alternativas de menor custo
	copyprop	Propagação das cópias das variáveis

Continua na próxima página

Tabela 5.1 – Passos de compilação do CoSy suportados pelo ambiente de compilação (continuação da página anterior).

Tipo	Nome	Descrição
Transformações de alto nível	condassigncreate	Substituição das dependências de controle por atribuições condicionais
	conevun	Avaliação de constantes e simplificação de expressões
	constprop	Propagação de constantes
	cse	Eliminação de subexpressões comuns
	demote	Redução da largura em <i>bits</i> das operações
	domorder	Ordenação de blocos com a ajuda da árvore dominadora
	exprprop	Propagação de expressões
	globcse	Eliminação de subexpressões comuns globais
	ifconvert	Substituição de dependências de controle por dependências de dados
	loopcanon	Ações de canonização em laços
	loopfuse	Fusão de dois laços adjacentes
	loopguard	Substituição dos <i>while-do</i> por <i>do-while</i> protegidos
	loophoist	Remoção de expressões dos laços.
	loopinvariant	Movimentação de códigos invariantes nos laços
	loopive	Eliminação da variável de indução do laço
	loopremove	Otimizador de laço vazio
	looprev	Reescrita do laço para contagem regressiva
	loopscalar	Substituição de escalares em laços
	loopstrength	Substituição de expressões de ponteiros por variáveis dentro dos laços
	lrrename	Renomeações de variáveis locais com base no seu tempo de vida
	misc	Otimizações diversas
	mvpostop	Movimentação das atribuições pós-operador
	noreturn	Remoção de código inatingível após não retornar a chamada da função
	promote	Divulgação do tamanho de expressão aritmética
	rodata	Dados somente de leitura
	scalarreplace	Substituição das variáveis estruturadas locais por escalares
	strength	Substituição de expressões por expressões de menor custo computacional
	tailmerge	Otimiza o tamanho do código através da combinação de caudas comuns (salto cruzado), sendo que a cauda é composta pelas últimas declarações de um bloco básico
	tailrec	Elimina a recursão de cauda a fim de reduzir o <i>overhead</i> das chamadas recursivas
	unroll	Desenrola o corpo de um laço (<i>loop</i>) de acordo com o fator especificado
	vprop	Avaliação e propagação de expressões
	vshrink	Redução no tamanho das variáveis locais
	vstrength	Substitui variáveis por outras de menor custo com base na análise do intervalo de valores
Transformações de baixo nível	dismemun	Reescrita das referências a <i>arrays</i> usando operações aritméticas de ponteiros
Transformações de baixo nível	hwhloopcreate	Substituição dos laços do código por construções em <i>hardware</i> (laços controlados por contadores)
	lowerbitfield	Reescrita das operações de <i>bits</i> através de expressões lógicas
	lowerboolval	Substituição das variáveis booleanas por atribuições em variável temporária do tipo inteiro para um fluxo de controle explícito
	lowerpfc	Conversão das chamadas à funções puras em chamadas de funções normais

No compilador ReflectC, algumas otimizações combinam dois ou mais passos do CoSy, executados em uma ordem fixa pré-estabelecida. Essa abordagem é geralmente adotada quando as transformações são muito sensíveis à ordem de aplicação (Kulkarni, 2007). No caso do ReflectC, essas otimizações compostas foram criadas para atender as recomendações do fabricante do compilador CoSy, o qual sugere fortemente a execução de outras transformações exatamente antes ou depois de um determinado passo de otimização (ACE, 2014). Por exemplo, o passo de compilação *loopstrength* do ReflectC compreende as transformações *loopanalysis*, *loopstrength* e *cleanloops*, as quais são executadas nessa ordem.

Nesta tese, o foco está na seleção e ordenação de passos de compilação e não na configuração de seus parâmetros internos. Assim, nosso espaço de otimizações original é formado por 49 passos de compilação do ReflectC, excluindo o passo *unroll*, o qual necessita da configuração adicional de parâmetros (identificação dos laços manipulados e seus respectivos fatores de desenrolamento), aumentando consideravelmente a complexidade do espaço de busca. O conjunto de otimizações adotado contém passos de análise, e transformações de alto e baixo nível. Intencionalmente, nossas abordagens de DSE exploram as sequências de otimização sem considerar qualquer conhecimento prévio sobre a ordem ou o estágio da compilação que elas costuma ser aplicadas.

5.6.3 Simulador

Durante o processo de exploração, a simulação ainda é bastante utilizada para avaliar o desempenho de uma possível solução no espaço de projeto para sistemas embarcados (Palermo et al, 2009).

Em nossas estratégias de DSE, o desempenho do código otimizado é estimado a partir de um simulador com precisão de ciclo desenvolvido pela ACE para o processador softcore Xilinx MicroBlaze (Xilinx, 2012). Ele permite simular duas versões do processador MicroBlaze: uma que usa 3 estágios de *pipeline* (busca, decodificação e execução) otimizada para redução da área usada no FPGA, e outra de 5 estágios de *pipeline* (busca, decodificação, execução, acesso à memória e armazenamento/*writeback*) otimizada para maximizar o desempenho (tempo de execução). Além do estágio de *pipeline*, o processador simulado também pode conter um circuito para deslocamento por ciclo de *bits* de uma palavra (*barrel shifter*), comumente usado na implementação em *hardware* de operações aritméticas com ponto-flutuante; um multiplicador e um divisor implementados em *hardware*; e uma instrução de comparação paralela (ex: PCMP). A escolha da configuração adotada pelo simulador é realizada através de parâmetros (*flags*) definidos na chamada do simulador.

O processador MicroBlaze usado em nossas simulações emprega 3 estágios de *pipeline* otimizados para área e contém todos os componentes de *hardware* disponíveis (*barrel shifter*, multiplicador, divisor e PCMP).

Capítulo 6

Experimentos

Diferentes estratégias de exploração do espaço de projeto (DSE) foram testadas durante esta pesquisa. O objetivo é avaliar a capacidade de encontrar sequências de otimização que propiciem a melhoria de desempenho das funções alvo quando executadas no processador utilizado. Apesar das principais abordagens serem baseadas na redução do espaço de busca através da seleção prévia de passos de compilação, algumas estratégias exploraram o espaço completo de busca, no qual são avaliadas todas as transformações disponíveis no ambiente de compilação. Os resultados dessas explorações são utilizados como parâmetro para a avaliação da eficiência e efetividade das estratégias de DSE que adotam a seleção baseada em agrupamento.

Outros experimentos também foram realizados com o intuito de investigar os efeitos de mudanças na forma de representação do código, nas métricas usadas no cálculo das matrizes de distância e no conjunto de referência sobre o desempenho da abordagem de exploração proposta.

Este capítulo descreve a configuração e metodologia de avaliação usadas nesses experimentos, bem como discute seus principais resultados.

6.1 Configuração do Ambiente dos Experimentos

Os experimentos foram realizados em um computador com um processador Intel Core2 Duo E7500 de 2.93GHz, 4 *Gbytes* de memória RAM, 150 *Gbytes* de disco rígido (HD) e sistema operacional Linux Ubuntu 13.10 64 *bits*.

Na implementação do nosso método de seleção baseado em agrupamento, o algoritmo *ncd* do pacote *complearn* (Cilibrasi et al, 2008) é usado para computar a matriz de distância, a partir das funções de referência e do código alvo. A matriz resultante é modificada para garantir sua simetria, através da média aritmética entre os pontos a_{ij} e a_{ji} , e adequar seu formato para o exigido pelo algoritmo de reconstrução de árvores filogenéticas. No processo de construção da árvore foi usado o algoritmo *neighbor* disponível no pacote

phylip (Shurman e Moehring, 1980). Esse algoritmo resulta em uma árvore no formato Newick, a qual é manipulada por nosso algoritmo de agrupamento implementado em C.

As estratégias de DSE avaliadas foram implementadas e executadas através do ambiente de compilação baseado em LARA, com o objetivo de obter as melhores sequências de otimização para 46 funções alvo, utilizadas como *benchmarks*, visando a execução em um processador *soft core* embarcado Xilinx MicroBlaze (Xilinx, 2012). Esse processador foi uma das arquiteturas alvo previamente empregadas no fluxo de projeto do REFLECT (Cardoso et al, 2013). Sua utilização em nossos experimentos permitiu o aproveitamento de toda a infraestrutura já existente no ambiente de compilação, em termos do *back-end* do ReflectC e do simulador com precisão de ciclos.

6.1.1 Processador MicroBlaze

O MicroBlaze (Xilinx, 2012) é um processador sintetizável de 32 *bits* com um conjunto reduzido de instruções (RISC - *Reduced Instruction Set Computer*), desenvolvido e mantido pela Xilinx. Esse processador foi projetado especificamente para operar, de forma otimizada, os circuitos dos FPGAs da empresa. Assim como outros processadores *soft cores*, ele permite a adequação de seus parâmetros de configuração aos requisitos do projeto. Uma visão geral da arquitetura desse processador é mostrada na Figura 6.1.

Figura 6.1: Diagrama de blocos do processador Xilinx Microblaze (Xilinx, 2012).

Seu conjunto de registradores é formado por 32 registradores de propósito geral de 32 *bits* e até 18 registradores de propósito específico. A arquitetura do conjunto de instruções (ISA - *Instructions Set Architecture*) é proprietária, formada por instruções de 32 *bits* de

3 operandos. Essas instruções podem ser de 2 tipos: instruções com 2 registradores de origem e um registrador de destino (tipo A), ou instruções com um registrador de origem, um operando imediato de 16 *bits* (expansível para 32 *bits*) e um registrador de destino (tipo B). A ISA do Microblaze inclui instruções para multiplicação e divisão implementadas em *hardware* para melhorar o desempenho da aplicação, desde que seja suportado pela arquitetura do FPGA utilizado. As instruções da ISA manipulam palavras, meia-palavras ou *bytes*, os quais podem estar nos formatos *big-endian* ou *little-endian* de acordo com o parâmetro de configuração adotado.

O sistema de barramento do MicroBlaze é formado, entre outros, por um barramento de memória local (LMB - *Local Memory Bus*), um barramento para comunicação rápida de uma via (FSL - *Fast Simplex Link bus*), e um barramento para acesso à memória externa proprietário, denominado *Xilinx CacheLink* (XCL). O LMB é um barramento síncrono usado para acesso aos blocos internos de memória RAM (*on-chip RAM*). Ele usa um conjunto mínimo de sinais de controle e um protocolo simples para garantir que a memória *on-chip* seja acessada em um único ciclo do relógio. O barramento FSL está disponível no nível de instrução e é usado para fornecer uma comunicação dedicada (ponto-a-ponto) e de baixa latência com os aceleradores em *hardware*. Essa abordagem é similar à adoção de instruções personalizadas para estender a ISA, mas tem a vantagem de tornar a velocidade global do *pipeline* do processador independente da função personalizada (Xilinx, 2012). O XCL foi projetado para permitir o acesso direto ao controlador de memória através de *buffers* FSL integrados, de modo a reduzir a latência e minimizar o número de instanciações necessárias.

A memória *cache* adota a arquitetura de *Harvard*, na qual existem caches distintas para instruções e dados. Ambas *caches* podem variar seu tamanho entre 2 e 64 *Kbytes*, formada por linhas de 4 ou 8 palavras. O acesso ao conteúdo da *cache* é feito através do mapeamento direto. A atualização da *cache* de dados pode ser realizada pelos métodos *write-through* ou *write-back*.

6.1.2 Conjunto de referência

Uma das inovações da abordagem proposta está na capacidade de selecionar um grupo promissor de passos de otimização a partir de sequências bem sucedidas nas funções de referência, ou seja, funções representativas escolhidas previamente a partir do histórico de funções já otimizadas/compiladas. Nosso processo não é baseado na construção de modelos e treinamentos (abordagem supervisionada) a fim de capturar o conhecimento intrínseco no histórico de funções (conjunto de referência), o que costuma demandar muito tempo e esforço computacional. Ao contrário, nossa abordagem é não supervisionada baseada na aplicação de técnicas de mineração de dados complexos sobre o conjunto de funções, formado pela nova função (alvo) e aquelas já otimizadas, a fim de identificar

padrões de similaridade entre os códigos, agrupando-os. A ideia básica é que códigos similares necessitam dos mesmos passos de compilação. Assim, as transformações usadas na exploração da função alvo são obtidas a partir das sequências das funções de referência presentes no mesmo grupo. O desempenho do agrupamento e, conseqüentemente, da exploração depende da qualidade do conjunto de referência adotado. As funções que formam esse conjunto devem ser representativas de modo a cobrir o máximo possível as diferentes regiões do espaço de funções.

Inicialmente, nosso conjunto de referência foi formado pelas funções descritas por Sanches et al (2011). Os códigos fontes dessas funções foram disponibilizados pelos autores, juntamente com as respectivas representações do DNA. Elas são variações das funções encontradas em diferentes repositórios de *benchmarks* (Guthaus et al, 2001)(Texas Instruments, 2003a)(Texas Instruments, 2003b)(to Verilog, 2014). Nesses códigos, as lógicas das funções foram inseridas na função *main()*, juntamente com os valores de inicialização de suas entradas, visando suas execuções e/ou simulações. Esse conjunto foi intencionalmente adotado por conter funções com diferentes características e que são empregadas em várias áreas de aplicação. As sequências de otimização que resultaram no melhor desempenho de cada função de referência foram adotadas como referência. Essas sequências de referência foram obtidas a partir dos experimentos realizados para a definição dos valores dos parâmetros de configuração do algoritmo genético, os quais estão descritos na Seção 6.3.

Entretanto, após os primeiros testes com o método de seleção baseado em agrupamento, esse conjunto inicial foi ligeiramente modificado, uma vez que esses testes nos permitiram identificar os seguintes fatos:

1. Duas funções de referência (*vecsum* e *dotprod*) não agregavam informação adicional ao agrupamento, uma vez que eles estavam sempre nos mesmos grupos e possuíam a mesma sequência de referência (*dismemun*, *cse*, *loopguard*, *strength*). Após a análise de seus códigos fonte, verificamos que os *loops* dessas funções são bem similares, justificando assim a proximidade nas árvores filogenéticas geradas.
2. Os passos de compilação necessários para a otimização de um grupo de funções alvo (*dsp_minval*, *dsp_maxval* e *dsp_maxidx*, que serão descritas a seguir) não estavam presentes nas sequências de referência de nenhuma das funções que compunham o conjunto inicial.

A partir dessas observações, optamos por trocar a função *vecsum* pela *dsp_minval*, a fim de tornar o conjunto de referência mais representativo. A Tabela 6.1 mostra as funções do conjunto de referência utilizado em nossos experimentos. São apresentados na tabela: o nome usado para identificar a função em nossos experimentos, a área de aplicação da função, e uma breve descrição de sua funcionalidade.

Tabela 6.1: Funções do conjunto de referência.

Função	Aplicação	Descrição
adpcm_code	Telecomunicações	Codificação de voz por ADPCM (<i>Adaptive Differential Pulse Code Modulation</i>)
adpcm_deco	Telecomunicações	Decodificação de voz por ADPCM
autcor	Processamento de sinais	Autocorrelação
bubble_sor	Computação	Ordenação de vetores pelo método bolha
dotprod	Matemática	Produto dos elementos do vetor
dsp_minval	Matemática	Menor elemento de um vetor
fdct	Processamento de imagens	Transformada discreta de cosseno (FDCT - <i>Forward Discrete Cosine Transform</i>)
fibonacci	Matemática	Série de <i>Fibonacci</i>
max	Matemática	Maior elemento de um vetor
pop_cnt	Computação	Contador de 1's em uma palavra binária
sobel	Processamento de imagens	Detecção de bordas pelo filtro Sobel

Embora os resultados obtidos pelas abordagens de DSE utilizando esse conjunto para o agrupamento e seleção dos passos de otimização tenham sido considerados satisfatórios, o impacto da adoção de outros conjuntos de referência são investigados na Seção 6.4.6.

6.1.3 Funções alvo (*benchmarks*)

Em um cenário ideal, durante a avaliação de um projeto de sistema embarcado, todas as aplicações que possam ser executadas em algum momento no sistema deveriam ser consideradas. Entretanto, esse cenário não existe, dado que é difícil mensurar o conjunto de aplicações que devem ser testadas, ou é impraticável, devido ao tempo que seria gasto para a execução desses testes. Assim, é necessária a adoção de um subconjunto de códigos que sejam representativos o suficiente para propiciar a avaliação, com a maior precisão possível, do desempenho dos sistemas e/ou dos métodos utilizados em seu projeto (que é o caso desta pesquisa). Os códigos que formam esse subconjunto são usualmente chamados de *benchmarks*. A maioria dos *benchmarks* incluem fragmentos de aplicações reais, ou códigos sintéticos que apresentam o mesmo comportamento dos algoritmos das aplicações reais.

Em nossos experimentos, as estratégias de DSE foram testadas sobre um conjunto de funções retiradas dos repositórios de *benchmark* da Texas Instruments (TI). A biblioteca DSP (TI C64x DSPLIB) (Texas Instruments, 2003a) é um repositório formado por funções DSP (*Digital Signal Processing*) usadas em aplicações de processamento digital de sinais. O repositório de processamento de imagens e vídeos (TI C64x IMGLIB) (Texas

Instruments, 2003b) fornece um conjunto de funções de alto desempenho que podem ser empregadas em uma ampla variedade de aplicações de processamento de vídeo e/ou imagem. As funções dessas bibliotecas são normalmente utilizadas em aplicações de tempo real com intensiva computação, nas quais o tempo de execução é crítico. Elas foram otimizadas para os dispositivos DSP (*Digital Signal Processor*) TMS320C64x da TI, de modo a atingir velocidades de execução consideravelmente maiores que seus respectivos códigos tradicionais. Para a exploração, essas funções são inseridas em programas completos, nos quais a função *main()* é usada para inicializar as variáveis de entrada e chamar a função alvo. Assim, é possível gerar os códigos que serão simulados.

Outros três códigos completam o conjunto das funções alvo usadas durante os testes do DSE. Um código implementa o sombreado de Gouraud (Gouraud, 1971), também conhecido como sombreado por interpolação de cor, muito usado em computação gráfica e que fazia parte do repositório de processamento de imagens e vídeos da TI, em uma versão anterior. Os outros dois (*autcor*— e *fdct*—) são variações das funções de referência *autcor* e *fdct*, com apenas uma modificação nos parâmetros de configuração. Na *autcor*— foi alterado o número de defasagens usado pelo algoritmo de 10 para 16. Na *fdct*— foi alterada a quantidade de transformadas discreta de cosseno (DCT - *Discrete Cosine Transform*) utilizadas de 10 para 4. Além dessas, a função *img_sobel*, que faz parte do repositório original de processamento de imagens e vídeos da TI, é uma variação bastante próxima de outra presente no conjunto de referência (*sobel*). Duas funções alvo (*img_sad_8x* e *img_mad_8x*) são variações de outras duas do mesmo conjunto de funções alvo (*img_sad_16* e *img_mad_16*), alterando-se apenas o tamanho do bloco manipulado pelas funções. Devido a essa similaridade, a utilização desses cinco códigos no conjunto de teste, formado por 46 funções diferentes, foi importante para verificar: (i) a capacidade do método de agrupamento em quantificar o grau de similaridade entre os códigos, e (ii) se a mudança nos parâmetros de configuração da função afeta as sequências de otimização resultantes do DSE. Como pode ser observado através das respectivas árvores filogenéticas apresentadas na Figura 6.2, o método agrupou adequadamente os pares de funções similares, de modo que eles sempre fossem os vizinhos mais próximos na topologia da árvore. Diferentes das demais funções alvo, os códigos de *goraud*—, *autcor*— e *fdct*— foram implementados diretamente na *main()*.

A Tabela 6.2 apresenta as funções retiradas do repositório de processamento digital de sinais (DSP), enquanto a Tabela 6.3 mostra as funções adotadas da biblioteca de processamento de imagens e vídeo. Juntas, elas totalizam as 46 funções da TI exploradas em nossos experimentos. A primeira coluna dessas tabelas apresenta os nomes usados para identificar cada função em nossos experimentos e seu nome original na biblioteca da TI (colocado entre parênteses). A segunda e terceira colunas indicam, respectivamente, o tipo do repositório (DSP ou IMG) e a categoria das funções conforme especificado nas bibliotecas (Texas Instruments, 2003a)(Texas Instruments, 2003b). A última coluna descreve

sucintamente a finalidade de cada função.

Os códigos fonte usados nesta tese, bem como as árvores filogenéticas, agrupamentos e os resultados gerados durante os experimentos estão disponibilizados *on-line* no sítio: <http://lcrserver.icmc.usp.br/projects/compilation-heuristics/repository>.

Figura 6.2: Árvores filogenéticas geradas a partir de códigos similares aos de algumas funções do conjunto de referência.

Tabela 6.2: Funções alvo retiradas do repositório de DSP da Texas Instruments.

Nome da função	Categoria	Descrição
autcor— (autcor)	Correlação	Autocorrelação
dsp_autoco (DSP_autoco)	Correlação	Autocorrelação
dsp_bexp_c (DSP_bexp)	Miscelânea	Máximo expoente de um vetor (por escala)
dsp_blk_e0 (DSP_blk_eswap16)	Miscelânea	Troca do tipo de <i>endians</i> (<i>endian-swap</i>) usado em um bloco de 16 <i>bits</i>
dsp_blk_e1 (DSP_blk_eswap32)	Miscelânea	Troca do tipo de <i>endians</i> (<i>endian-swap</i>) usado em um bloco de 32 <i>bits</i>
dsp_blk_e2 (DSP_blk_eswap64)	Miscelânea	Troca do tipo de <i>endians</i> (<i>endian-swap</i>) usado em um bloco de 64 <i>bits</i>
dsp_blk_mo (DSP_blk_move)	Miscelânea	Movimentação de um bloco da memória
dsp_dotpro (DSP_dotprod)	Matemática	Produto dos elementos do vetor
dsp_dotp_s (DSP_dotp_sqr)	Matemática	Produto e quadrado dos elementos do vetor
dsp_fir_cp (DSP_fir_cplx)	Filtragem e convolução	Filtro FIR complexo
dsp_firlms (DSP_firlms2)	Filtragem adaptativa	Filtro FIR baseado na filtragem adaptativa pela média dos mínimos quadrados (LMS - <i>Least Mean Squares</i>)
dsp_fltoq1 (DSP_fltoq15)	Miscelânea	Conversão de <i>float</i> para o formato Q15 (ponto fixo)
dsp_mat_mu (DSP_mat_mul)	Matriz	Multiplicação de matriz
dsp_mat_tr (DSP_mat_trans)	Matriz	Transposição de matriz
dsp_maxidx (DSP_maxidx)	Matemática	Índice do maior elemento de um vetor
dsp_maxval (DSP_maxval)	Matemática	Maior elemento de um vetor
dsp_minerr (DSP_minerror)	Miscelânea	Busca pelo erro de energia mínimo
dsp_mul32 (DSP_mul32)	Matemática	Multiplicação de um vetor de 32 <i>bits</i>
dsp_neg32 (DSP_neg32)	Matemática	Negação dos elementos de um vetor de 32 <i>bits</i>
dsp_q15tof (DSP_q15tofl)	Miscelânea	Conversão do formato Q15 (ponto fixo) para <i>float</i>
dsp_vecsum (DSP_vecsumsq)	Matemática	Soma dos quadrados de um vetor
dsp_w_vec (DSP_w_vec)	Matemática	Soma ponderada de um vetor

Tabela 6.3: Funções alvo retiradas do repositório de processamento de vídeo e imagens da Texas Instruments.

Nome da função	Categoria	Descrição
fdct—— (fdct)	Compressão / descompressão	Transformada discreta de cosseno
gouraud— (gouraud)	Gráficos 3D	Sombreamento de Gouraud
img_bounda (IMG_boundary)	Análise de Imagem	Operador estrutural de fronteira (borda)
img_conv_3 (IMG_conv_3x3)	Filtragens de figuras / conversão de formatos	Convolução 3x3
img_corr_g (IMG_corr_gen)	Filtragens de figuras / conversão de formatos	Correlação generalizada
img_dilate (IMG_dilate_bin)	Análise de Imagem	Dilatação binária 3x3
img_erode_ (IMG_erode)	Análise de Imagem	Erosão binária 3x3
img_fdct_8 (IMG_fdct_8x8)	Compressão / descompressão	Transformada discreta de cosseno
img_idct_8 (IMG_idct_8x8)	Compressão / descompressão	Inversa da transformada discreta de cosseno (IDCT - <i>Inverse Discrete Cosine Transform</i>)
img_mad_8x (IMG_mad_8x8)	Compressão / descompressão	Diferença absoluta mínima em um bloco 8x8
img_mad_16 (IMG_mad_16x16)	Compressão / descompressão	Diferença absoluta mínima em um bloco 16x16
img_median (IMG_median_3x3)	Filtragens de figuras / conversão de formatos	Filtro mediano 3x3
img_perime (IMG_perimeter)	Análise de Imagem	Operador estrutural de perímetro
img_pix_ex (IMG_pix_expand)	Filtragens de figuras / conversão de formatos	Expansão de <i>Pixel</i>
img_pix_sa (IMG_pix_sat)	Filtragens de figuras / conversão de formatos	Saturação de <i>Pixel</i>
img_quanti (IMG_quantize)	Compressão / descompressão	Quantização de matrizes com arredondamento
img_sad_8x (IMG_sad_8x8)	Compressão / descompressão	Soma das diferenças absolutas em um bloco 8x8
img_sad_16 (IMG_sad_16x16)	Compressão / descompressão	Soma das diferenças absolutas em um bloco 16x16
img_sobel_ (IMG_sobel)	Análise de Imagem	Detecção de borda pelo filtro Sobel
img_wave_h (IMG_wave_horz)	Compressão / descompressão	Transformada <i>wavelet</i> horizontal
img_wave_v (IMG_wave_vert)	Compressão / descompressão	Transformada <i>wavelet</i> vertical
img_yc_de0 (IMG_yc_demux_be16)	Filtragens de figuras / conversão de formatos	Demultiplexação YCbCr (fonte <i>big endian</i>)
img_yc_de1 (IMG_yc_demux_le16)	Filtragens de figuras / conversão de formatos	Demultiplexação YCbCr (fonte <i>little endian</i>)
img_ycbcr4 (IMG_ycbcr422p_rgb565)	Filtragens de figuras / conversão de formatos	Conversão do espaço de cores RGB 5:6:5 para YCbCr 4:2:2/4:2:0 aplanado

6.2 Metodologia de Avaliação

Durante a exploração do espaço de soluções, cada sequência candidata é avaliada de modo que o algoritmo possa minimizar a latência do código, ao mesmo tempo que tenta minimizar o tamanho da sequência de otimização adotada. O processo de avaliação de uma sequência se inicia com a aplicação da mesma na otimização do código fonte original através do ReflectC. O código otimizado é então compilado por um compilador da linguagem C para MicroBlaze (`microblazecc`), gerando o código executável. O simulador `microblazesim` executa esse código e gera um relatório de desempenho, do qual é extraída a informação de latência da execução (dada em número de ciclos do relógio). Na etapa de otimização podem ocorrer alguns erros decorrentes da interação entre os passos de compilação (ordenação imprópria). Esses casos são tratados pelo algoritmo de avaliação e um valor muito alto é atribuído para a latência de modo que a solução seja descartada.

A latência permite a avaliação das soluções (sequências) para a função específica na qual ela foi aplicada. Os valores de latência são altamente dependentes do código da função (tamanho e tipo de instruções) e dos argumentos de entrada usados durante a simulação. A fim de realizar uma avaliação comparativa entre as diferentes abordagens aplicadas sobre o conjunto completo das funções alvo, o *speedup* foi adotado como a métrica de avaliação do desempenho de uma estratégia de DSE. Assim, a cada sequência avaliada sobre uma função específica, após o cálculo da latência, o valor do *speedup* também é obtido, através da razão entre a latência gerada pelo código original (sem otimizações) e a latência resultante do código otimizado pela sequência em análise. Essa razão permite realizar a comparação mesmo entre funções que apresentam latências com ordens de grandeza diferentes. O tamanho da sequência é usado como critério de desempate quando duas sequências resultam na mesma latência. Ao final do DSE, o algoritmo retorna a melhor sequência obtida, juntamente com os valores da latência calculada, do *speedup* alcançado e do tempo de exploração (dado em segundos).

Em nossos experimentos, a avaliação e comparação dos desempenhos das estratégias de DSE, é realizada considerando-se o desempenho médio das estratégias em todo o conjunto de 46 funções alvo. Os valores médios de *speedup* e do tempo de execução da exploração são obtidos através da média geométrica dos resultados individuais de cada função alvo. O uso da média geométrica visa destacar a tendência central do conjunto de funções otimizadas através da normalização dos seus valores individuais. Dessa forma, qualquer mudança percentual em um dos valores individuais provoca um efeito similar no valor médio gerado. A média geométrica é mais adequada para tratar valores normalizados ou percentuais, como os *speedups*, pois é menos afetada pela normalização que a média aritmética (Fleming e Wallace, 1986). Além disso, a média geométrica é indicada para lidar com conjuntos que apresentam valores com diferentes ordens de grandeza, como no caso da comparação dos tempos de execução, que apresenta uma grande variação de uma

função para outra.

Formalmente, a média geométrica (μ_{geom}) de um conjunto $C = \{a_1, a_2, a_3, \dots, a_n\}$ é dada por:

$$\mu_{geom}(C) = \left(\prod_{i=1}^n a_i \right)^{\frac{1}{n}} = \sqrt[n]{a_1 \times a_2 \times a_3 \times \dots \times a_n} \quad (6.1)$$

Com o objetivo de avaliar a eficiência da abordagem proposta, comparamos o desempenho das estratégias de DSE que utilizam o método de seleção baseado em agrupamento com aqueles obtidos pelas explorações realizadas sobre todo o espaço de otimizações. Nas explorações completas (DSE sobre todo o espaço de otimizações) foram empregados os algoritmos de busca retirados da literatura; são eles: algoritmo genético (AG) (Goldberg, 1989), amostragem aleatória e algoritmo de inserção (Huang et al, 2013).

Técnicas de busca inteligentes como os AGs costumam alcançar ganhos bem próximos do desempenho ótimo (Kulkarni et al, 2007)(Jantz e Kulkarni, 2013). Além disso, o uso da estratégia de DSE baseada em AG (DSE-AG) resultou nos melhores valores de latência/*speedup* para todas as funções avaliadas. Então, no escopo deste trabalho, assumimos que os resultados do DSE-AG correspondem aos melhores valores que poderiam ser alcançados (menor latência e maior *speedup*), tanto no espaço completo quanto no reduzido. Assim, a eficiência das demais estratégias de DSE é verificada em função da proximidade do seu desempenho com aquele obtido pelo DSE-AG.

Assim, a eficiência da abordagem é verificada em função da sua proximidade de seus resultados com aqueles obtidos pelo AG sobre o conjunto completo de otimizações.

Os resultados obtidos para todas as estratégias testadas são apresentados na Seção 6.4.

6.3 Configuração dos Parâmetros do Algoritmo Genético

A configuração inicial dos parâmetros do algoritmo genético (AG) foi definida a partir de experimentos realizados sobre um código que implementa o filtro *Subband*. O filtro *Subband* é usado na compressão de dados para sinais de áudio e de vídeo (codificador de áudio MPEG), e serve para decompor um sinal em diferentes bandas de frequência, codificando-as independentemente. O código foi desenvolvido pela Coreworks e foi usado no escopo do Projeto Reflect para a avaliação do ambiente de compilação. Ele foi escolhido por ser mais complexo que as funções da Texas Instruments, e já possuir uma latência ótima previamente conhecida, a qual foi usada como parâmetro no processo de refinamento.

Neste contexto, cada configuração do AG foi executada 5 vezes sobre o espaço completo de otimizações e selecionada aquela que garantiu o valor ótimo em todas as execuções com o menor tempo médio de exploração. As configurações testadas foram geradas

a partir da combinação de diferentes tamanhos de população (100, 200, 300, 400 e 500), taxas de *crossover* (60, 70 e 80) e taxas de mutação (30, 40 e 50). Todas as execuções consideraram sequências de no máximo 32 passos de compilação, uma vez que a partir de 21 transformações já é possível atingir a latência mínima. Dentre as configurações testadas, o melhor desempenho foi obtido com o AG executado sobre uma população de 300 indivíduos e os operadores de *crossover* e mutação aplicados com uma probabilidade de 60% e 40%, respectivamente.

Essa configuração foi então aplicada sobre as funções usadas na pesquisa de Sanches et al (2011), cujo os códigos foram disponibilizados pelos autores, para avaliar a eficiência da exploração baseada em AG, bem como determinar qual é o tamanho máximo ideal para as sequências exploradas, visando obter o melhor compromisso entre a precisão do método e o tempo de exploração. Nesse processo foram geradas sequências com 4, 8, 16 ou 32 passos de compilação, a fim de definir qual é o tamanho máximo ideal para garantir uma boa otimização para as funções manipuladas.

Conforme pode ser observado no gráfico da Figura 6.3, não existe um ganho de desempenho significativo entre sequências de 16 e 32 transformações que justifique um aumento de aproximadamente 60% no tempo médio de exploração (de 5509 para 8749 segundos). A *adcpm_code* foi a única função que melhorou o desempenho com a adoção de sequências com 32 otimizações (1,27 para 1,29). Assim, todos os experimentos foram executados com o tamanho máximo de 16 passos de compilação.

Figura 6.3: Evolução da média geométrica do *speedup* e do tempo de execução em função do tamanho das sequências usada pelo AG.

A abordagem baseada em AG é empregada para explorar tanto o conjunto completo de otimizações, quanto o conjunto reduzido pelo nosso método de seleção. O tamanho de população igual a 300 foi determinado empiricamente para o espaço de busca completo.

Devido à grande diferença de cardinalidade dos espaços gerados pelo dois conjuntos, foi necessário determinar qual o tamanho de população que poderia ser usado na exploração do espaço reduzido sem comprometer os resultados obtidos. Neste contexto, experimentos exploratórios utilizando as funções alvo e o conjunto reduzido mostraram que é suficiente utilizar uma população de 30 indivíduos no AG para atingir os mesmos *speedups* obtidos caso uma população de 300 indivíduos fosse utilizada, com um tempo de exploração $9\times$ menor (412 contra 3828 segundos).

6.4 Resultados Experimentais

Nesta tese, foram realizados vários experimentos com o objetivo de avaliar a eficiência da nossa abordagem em explorar boas sequências de otimização para as funções alvo, bem como investigar o efeito de alterações na configuração do método de seleção na qualidade dos resultados obtidos. Os resultados desses experimentos são apresentados a seguir.

6.4.1 Avaliação das estratégias de DSE

Durante a pesquisa, uma variedade de estratégias de busca foram usadas para explorar o espaço de otimizações, resultando em diferentes precisões e tempos de execução, com o objetivo de verificar a eficácia da abordagem de seleção de passos de otimização baseada em agrupamento para o processo DSE. Essas estratégias diferem basicamente na técnica e no espaço de busca empregados na pesquisa das melhores sequências de otimização para uma dada função.

Os experimentos sem a seleção prévia das otimizações, chamadas de DSE completo, foram executados sobre um espaço de busca formado por 49 passos de otimização do compilador ReflectC (Tabela 5.1). Dado que em nossos experimentos são geradas sequências com no máximo 16 passos de otimização, o tamanho do espaço de busca completo (ρ) pode ser calculado pela Equação 2.4 e sua ordem de grandeza é dada pela desigualdade $49^{16} < \rho < 50^{16}$. Portanto, sua exploração exaustiva não é possível na prática. A exploração que emprega nossa abordagem de seleção baseada em agrupamento, denominada DSE reduzido, utiliza um subconjunto de otimizações escolhidas a partir da similaridade entre o código alvo e as funções de referência. A redução no conjunto de otimizações varia de acordo com a função alvo explorada, sendo que o menor número de passos obtido é igual 5 para a função *autcor*— e o maior número foi 20 para a função *img_bounda*. Considerando que na média esse subconjunto é formado por 13 otimizações, a ordem de grandeza do espaço de busca resultante obedece a relação $13^{16} < \rho < 14^{16}$, ou seja, bem menor que o espaço original. Entretanto, ainda assim é necessária a adoção de algum método de busca capaz de explorá-lo adequadamente.

Para a melhor compreensão de nossas análises, os resultados desses experimentos serão apresentados em 3 etapas. Na primeira, discutiremos os resultados obtidos através das estratégias de DSE baseadas nos métodos individuais de busca (amostragem aleatória, inserção e limpeza), os quais serão comparados com o desempenho do DSE efetuado pelo algoritmo genético, tanto no espaço completo, quanto no reduzido. Na segunda etapa, discutiremos os resultados obtidos através das estratégias de DSE baseadas nos métodos híbridos de busca (*Limpeza & Inserção* - CI e *Inserção & Limpeza* - IC), que serão comparados com o desempenho do DSE-AG no espaço completo, além de uma sequência fixa que atinge a melhor otimização, considerando-se todas as funções de referência. Finalmente, serão apresentadas algumas considerações adicionais sobre o desempenho alcançado tanto nas abordagens baseadas em métodos individuais, quanto naquelas que adotam os métodos híbridos.

Métodos individuais de busca

Estratégias de DSE baseadas em amostragem aleatória (DSE-RS), algoritmo genético (DSE-AG) e algoritmo de inserção (DSE-Ins) foram executadas nos dois espaços (completo e reduzido). A abordagem baseada no algoritmo de limpeza (DSE-Limp) foi empregada apenas sobre o espaço reduzido dado que é fortemente influenciada pela ordem dos passos na sequência inicial e nosso método não emprega nenhuma análise preliminar para estabelecer, de forma adequada, essa ordem para o conjunto completo de passos de otimização.

O desempenho das estratégias que adotam o DSE reduzido é comparado com aquele alcançado pelas estratégias que empregam o DSE sobre o espaço completo. Supondo que a abordagem baseada em AG geralmente resulta em valores próximos do ótimo (Kulkarni et al, 2007)(Jantz e Kulkarni, 2013), os *speedups* alcançados pelo DSE-AG completo são utilizados como meta para as outras abordagens de exploração. Nesse caso, a eficiência da abordagem é definida em função da proximidade de seus resultados com aqueles obtidos pelo DSE-AG.

A Figura 6.4 mostra os melhores *speedups* alcançados por cada uma das estratégias de DSE baseadas em um único algoritmo de busca (abordagens não híbridas) para as 46 funções alvo. Para facilitar a comparação entre as abordagens, a média geométrica dos melhores *speedups* individuais também é apresentada na última coluna do gráfico.

Todas as estratégias de exploração propiciaram uma melhoria no desempenho em relação ao código original (sem otimização). Esses ganhos de desempenho estão associados a diversas mudanças causadas pela aplicação dos passos de otimização, tais como: redução do número de instruções executadas, melhoria na predição de saltos em função do rearranjo do código, entre outros.

Figura 6.4: Melhor *speedup* obtido para cada função alvo usando estratégias de DSE baseadas em um único algoritmo de busca sobre os espaços reduzido e completo.

No geral, as explorações que utilizam o espaço completo (representadas por linhas no gráfico) obtiveram um *speedup* médio maior que suas equivalentes no espaço de otimizações reduzido (representadas por barras verticais). Após a análise das sequências obtidas, observamos alguns fatos que contribuem para esta diferença: (i) algumas funções precisam de otimizações específicas, as quais não estão presentes nas sequências de otimização das funções de referência utilizadas (conjunto de referência), como por exemplo, a função *img_ycbcr4* que precisa dos passos *mvpostop* e *globcse*; (ii) alguns passos de otimização são usados por poucas funções de referência (1 ou 2), dificultando sua presença no subconjunto gerado (ex: *loopfuse* e *cache*); e (iii) os algoritmos simples têm dificuldade em gerar sequências com repetições de passos, o que é necessário na otimização das funções *dsp_bexp_c* e *dsp_fir_cp*.

Como esperado, as estratégias de DSE baseadas em AG alcançaram os melhores *speedups* em ambos os espaços de busca, sendo que o uso do AG sobre o espaço completo foi a melhor estratégia avaliada, resultando nos maiores valores de *speedup* para todas as 46 funções alvo. As estratégias baseadas em amostragem aleatória atingiram *speedups* próximos aos obtidos pelas respectivas explorações com o AG. Entretanto, para isso, elas precisam avaliar um conjunto $10\times$ maior de soluções que aquele explorado pelo AG em ambos os espaços. As estratégias que adotam o algoritmo de inserção tiveram os piores resultados para os espaços completo e reduzido, sendo esse último o pior dentre todas as abordagens. O algoritmo de inserção apresentou dificuldades em encontrar boas sequências para funções que outras abordagens conseguiram otimizar tanto no espaço completo quanto reduzido. Por exemplo, ambas as abordagens não foram capazes de encontrar a melhor sequência de otimização para as funções *autcor*—, *dsp_maxidx* e *dsp_maxval*. Em contrapartida, na exploração sobre o espaço completo, ele foi capaz de encontrar sequências melhores que as demais abordagens (exceto o AG completo) para as funções *dsp_fir_cp* e *img_fdct_8*. Acreditamos que esse comportamento é influenciado pelas interações entre os passos de otimização. Por ser um algoritmo guloso, a decisão de manutenção de uma nova otimização na sequência é baseada no estado atual da solução (composição e desempenho correntes), e influenciadas pela ordem com a qual os passos de otimização são manipulados. Assim, o algoritmo é suscetível a mínimos locais que podem impedir a obtenção do ótimo global. O DSE baseado no algoritmo de limpeza foi a estratégia de exploração mais simples e rápida utilizada em nossa pesquisa. Ela também é bastante afetada pela ordem inicial da sequência e pelas interações entre as otimizações durante a construção da solução. Bons passos de otimização (encontrados nas soluções de outras abordagens) são removidos da solução por não resultarem na melhoria do *speedup* em relação à solução corrente. Por outro lado, essa estratégia simplista foi capaz de alcançar *speedups* muito próximos das estratégias de DSE sobre o espaço reduzido baseadas em algoritmo genético e amostragem aleatória e do algoritmo de inserção usado no espaço completo. Além disso, o DSE-Limp apresentou uma diferença de apenas 0,1 no

speedup médio em relação ao DSE-AG completo (considerado como meta).

Métodos híbridos de busca

Nestes experimentos, duas abordagens híbridas que combinam os algoritmos de limpeza e inserção foram implementadas, considerando o comportamento observado na exploração a partir desses algoritmos individualmente. A abordagem de *Limpeza & Inserção* (DSE-CI) aplica o algoritmo de limpeza sobre o espaço reduzido e a sequência resultante é usada como configuração inicial para o algoritmo de inserção, o qual realiza uma única passagem no espaço completo de otimizações. A abordagem de *Inserção & Limpeza* (DSE-IC) aplica uma única iteração do DSE-Ins completo sobre a sequência decorrente do espaço reduzido, sendo seu resultado submetido ao algoritmo de limpeza. O objetivo das abordagens híbridas é superar as limitações dos algoritmos originais, propiciando uma exploração mais eficaz do espaço de busca. A ideia é utilizar o espaço completo para refinar o resultado obtido a partir do conjunto reduzido. Assim, os métodos são híbridos por misturarem operações de duas estratégias diferentes (inserção e limpeza), mas também são híbridos por trabalharem tanto com o espaço reduzido, que gera uma sequência inicial, quanto com o espaço completo, que refina essa sequência.

Além disso, tal como na abordagem adotada por Kulkarni e Cavazos (2012), realizamos uma exploração do espaço completo de otimizações a fim de encontrar uma única sequência de otimização que resulte no melhor desempenho global para as 11 funções do conjunto de referência. A ideia é gerar uma sequência de otimização representativa, como uma espécie de "-O2" para o compilador utilizado, a qual é usada para comparar as sequências especializadas. Durante o processo, cada sequência potencial foi avaliada sobre todas as funções de referência e a média geométrica foi calculada a partir de seus *speedups* em relação ao código original (sem otimizações). A sequência resultante, a qual denominamos de "melhor sequência global", é formada por 16 passos de otimização, na seguinte ordem: {*loopstrength*, *vstrength*, *lrrename*, *loopprev*, *loopive*, *lowerboolval*, *funceval*, *setpurity*, *loopstrength*, *chainflow*, *loopbcount*, *vprop*, *loopscalar*, *dismemun*, *cse*, *strength*}. Ela foi então aplicada sobre as funções alvo analisadas com o objetivo de validar o nível de representatividade do conjunto de referência.

Os melhores *speedups* individuais (para cada função alvo) atingidos pelas estratégias de DSE baseadas nas abordagens híbridas e com a melhor sequência "global" obtida sobre todo o conjunto de referência foram comparados com o DSE-AG sobre o espaço completo. A Figura 6.5 mostra a comparação dos valores individuais, bem como da média geométrica dos *speedups* resultantes em cada abordagem.

Figura 6.5: *Speedups* obtidos para cada função alvo usando estratégias de DSE baseadas em algoritmos híbridos e a sequência de compilação que melhor otimizou todas as funções do conjunto de referência.

As abordagens híbridas alcançaram *speedups* muito próximos àqueles gerados pelo DSE baseado em AG. A partir da comparação entre as sequências geradas pelo AG e pelas abordagens híbridas para as funções *dsp_bexp_c* e *dsp_fir_c*, evidenciou-se a dificuldade dos métodos simples de busca em gerar sequências com repetições de otimizações. Mesmo o DSE-AG completo apresentou dificuldade em encontrar as repetições nessa sequência; em 1/5 das execuções, o AG não foi capaz de encontrar a sequência com repetição.

A adoção da melhor sequência "global" resultou em *speedups* próximos aos do DSE sobre o conjunto reduzido, embora seus valores sejam piores que os das outras abordagens, por exemplo, quando se explora sequências de otimização para a função *img_conv_3*.

A Figura 6.6 mostra a quantidade de passos de otimização distintos existentes no espaço de busca inicialmente explorado pelas abordagens híbridas, assim como o número de otimizações contidas nas melhores sequências alcançadas pelo algoritmo genético completo e pelas abordagens híbridas (DSE-CI e DSE-IC) para cada função avaliada. A última coluna do gráfico apresenta a média aritmética para auxiliar na comparação entre as abordagens de DSE.

sequências das funções *dsp_bexp_c* e *dsp_fir_cp* utilizam, respectivamente, 5 e 7 passos distintos. O restante das 16 otimizações são repetições dos passos de otimização *loopguard* e *loopcanon*. Também foi verificado que as sequências finais geradas em cada algoritmo (AG, IC e CI) são muito similares para várias funções. De fato, os algoritmos híbridos atingiram um desempenho semelhante ao da exploração do espaço completo pelo AG.

Análise da eficácia das estratégias de DSE

Foi possível verificar que nossa abordagem é capaz de encontrar boas sequências de otimização. Entretanto, existem diferenças de desempenho entre as estratégias de DSE avaliadas e, principalmente, entre os espaços de busca explorados, dado que o espaço reduzido é diretamente influenciado pelo conjunto de funções usado como referência. Neste contexto, mesmo nossa estratégia mais simples (DSE-Limp) foi capaz de obter *speedups* similares aos obtidos pela melhor sequência "global" gerada a partir do nosso conjunto de referência. Também foi possível verificar que nossas abordagens híbridas superam as deficiências apresentadas pelos métodos simples de busca individualmente, obtendo um desempenho próximo ao AG no espaço completo. Tal observação é corroborada através da média geométrica dos melhores *speedups* alcançados para cada estratégia de DSE, como destacado na Figura 6.7.

Figura 6.7: Média geométrica dos melhores *speedups* de acordo com a estratégia de DSE utilizada.

O *speedup* médio obtido pela melhor sequência "global" indica o grau de represen-

tatividade do conjunto de referência adotado. O fato da nossa abordagem mais simples (DSE-Limp) atingir o mesmo valor de *speedup* médio, nos indica que ela pode ser usada quando os requisitos e limitações não são tão rígidos, ou ainda para gerar sequências de otimização com desempenho similar às aquelas mais generalistas resultantes dos compiladores tradicionais.

Basicamente, as abordagens podem ser agrupadas em 3 grupos de acordo com seus *speedups* médios. O primeiro grupo é formado pelo DSE usando algoritmo genético (43%), as abordagens híbridas DSE-CI (42%) e DSE-IC (41%) e a amostragem aleatório a partir do espaço completo (40%). O grupo intermediário compreende o algoritmo de inserção aplicado sobre o espaço completo (35%); as estratégias de DSE sobre o espaço reduzido baseadas no algoritmo genético e na amostragem aleatória (ambos com 34%), e no algoritmo de limpeza (33%); e a melhor sequência "global" (33%). O último grupo é formado apenas pelo DSE do algoritmo de inserção aplicado sobre o espaço reduzido (27%). Note que a diferença entre as médias geométricas dos grupos é em torno de 0,16.

Para evidenciar ainda mais as diferenças de desempenho entre as abordagens, foi realizada uma comparação refinada, na qual é analisado apenas um subconjunto das funções. A escolha desse subconjunto é baseada na diferença (*Dif*) entre a melhor latência, obtida pelo AG, e a média dos valores das latências retornadas por todas as demais abordagens de DSE. A ideia é selecionar apenas as funções que, no geral, foram mais difíceis de otimizar, removendo aquelas cuja melhor latência é obtida em quase todas as estratégias de DSE. Os resultados apresentados na Tabela 6.4 demonstram que as diferenças entre as estratégias de DSE ficaram mais evidentes com a filtragem e diminuí à medida que novas funções são consideradas. Além do conjunto completo (46 funções), também são apresentados os valores de *speedup* para 3 subconjuntos, gerados de acordo com o grau de dificuldade para a otimização (definido por *Dif*). As funções presentes em cada subconjunto são listadas a seguir:

- **Grupo de 18 funções** ($G_{18} \Leftrightarrow Dif \geq 0,05$) = { *dsp_blk_e0*, *dsp_fir_cp*, *dsp_mat_mu*, *dsp_mat_tr*, *dsp_maxidx*, *dsp_maxval*, *dsp_minerr*, *fdct*—, *goraund*—, *img_bounda*, *img_dilate*, *img_erode*, *img_fdct_8*, *img_idct_8*, *img_mad_16*, *imd_mad_8x*, *img_pix_sa*, *img_sad_16* }
- **Grupo de 24 funções** ($G_{24} \Leftrightarrow Dif \geq 0,03$) = $G_{18} \cup$ { *dsp_blk_e1*, *img_conv_3*, *img_corr_g*, *img_sad_8x*, *img_sobel*—, *img_ycbcr4* }
- **Grupo de 30 funções** ($G_{30} \Leftrightarrow Dif \geq 0,01$) = $G_{24} \cup$ { *autcor*—, *dsp_autoco*, *dsp_blk_e2*, *dsp_blk_mo*, *img_median*, *img_yc_de1* }

As linhas da tabela estão ordenadas de acordo com o desempenho obtido no subconjunto mais restrito (18 funções). Além disso, a maior média geométrica dos melhores *speedups* alcançada em cada subconjunto está destacada em negrito.

Tabela 6.4: Desempenho das estratégias de DSE de acordo com o número de funções.

Estratégia	18	24	30	46
DSE-AG completo	1,65	1,56	1,49	1,43
DSE-CI	1,63	1,54	1,48	1,42
DSE-IC	1,62	1,53	1,47	1,41
DSE-RS completo	1,58	1,50	1,45	1,40
DSE-AG reduzido	1,45	1,39	1,36	1,34
DSE-RS reduzido	1,45	1,39	1,36	1,34
DSE-Limp	1,45	1,38	1,35	1,33
DSE-Ins completo	1,44	1,41	1,37	1,35
Melhor seq. "global"	1,44	1,37	1,34	1,33
DSE-Ins reduzido	1,29	1,27	1,26	1,27

No geral, o comportamento entre as abordagens foi mantido. Exceto pelo DSE-Ins completo, o qual apresentou resultados inferiores aos das abordagens sobre o espaço reduzido para o subconjunto de 18 funções. Entretanto, a diferença entre o AG e as estratégias de DSE sobre o conjunto reduzido é realçado em função do nível de filtragem adotado. As abordagens híbridas permaneceram estáveis e mantiveram seus *speedups* médios próximos ao DSE-AG completo. Com exceção da exploração do espaço reduzido baseada no algoritmo de inserção (DSE-Ins reduzido), todas as demais estratégias de DSE resultaram em *speedups* médios iguais ou superiores à melhor sequência "global", confirmando que essa abordagem pode ser usada em otimizações com expectativa de desempenho próximo ao obtido por estratégias mais generalizadas, como as adotadas pelos compiladores comerciais (ex: GCC -O). Entretanto, o uso das abordagens de DSE permite lidar com novas otimizações sem a necessidade do estudo prévio de suas iterações, usualmente realizados no aprimoramento das regras "-O". Tal característica é importante no contexto dos compiladores modernos, os quais são constantemente atualizados para incluir novos passos de otimização. Em contrapartida, esse comportamento também demonstra que a qualidade dos resultados obtidos pelas estratégias de DSE sobre o conjunto reduzido são influenciados pela representatividade do conjunto de referência adotado.

Dependendo dos requisitos impostos no projeto, uma redução em termos de *speedup* pode ser aceitável, especialmente se considerarmos os ganhos no tempo de execução do DSE. A Figura 6.8 apresenta o tempo de execução das estratégias de DSE para cada função alvo avaliada. A última coluna do gráfico mostra o tempo médio de execução para cada estratégia utilizada na exploração. Boa parte desse tempo de execução é usado na avaliação das sequências de otimização exploradas, ou seja, na geração do código otimizado e na sua simulação. A diferença de tempo entre as funções alvo está relacionada com a complexidade da função e, conseqüentemente, com o tempo necessário para a sua simulação. A diferença de tempo entre as diferentes estratégias de DSE está diretamente associada à quantidade de pontos explorados no espaço de busca.

Figura 6.8: Tempo de execução das estratégias de DSE no espaço completo ou reduzido obtidos para cada função alvo.

Existem grandes diferenças na ordem de grandeza dos tempos de execução do DSE entre as abordagens de exploração do espaço completo baseadas em AG e amostragem aleatória e as demais estratégias de DSE, forçando-nos a adotar duas escalas diferentes para representar esses valores no gráfico. A escala da direita é usada para representar os tempos médios de execução das estratégias mais demoradas (DSE-AG e DSE-RS completos), enquanto que a escala da esquerda representa as abordagens híbridas (DSE-CI e DSE-IC), todas as estratégias de exploração do espaço reduzido, e o DSE baseado no algoritmo de inserção sobre o espaço completo (DSE-Ins completo). Embora o DSE-AG sempre alcance sequências iguais ou melhores de otimização, ele também gasta mais tempo de execução durante o DSE. Entre as abordagens de exploração do espaço reduzido, o algoritmo de limpeza é a mais rápida e a amostragem aleatória é a mais lenta. Essa diferença está associada à quantidade de soluções que precisam ser avaliadas por cada abordagem para a obtenção do desempenho relatado. A Figura 6.9 apresenta o tempo médio de execução de cada estratégia de DSE obtido a partir da média geométrica dos valores individuais das funções exploradas.

Figura 6.9: Média geométrica do tempo de execução das estratégias de DSE.

Como pode ser observado, melhorias significativas foram obtidas com a redução do espaço em relação ao tempo de execução quando comparamos com as explorações no espaço completo. Entre as abordagens realizadas sobre o espaço completo, o algoritmo de inserção foi o mais rápido, apresentando um tempo médio de execução inferior ao da abordagem híbrida DSE-IC, além de ser um pouco superior aos das abordagens DSE-AG e DSE-RS sobre o conjunto reduzido. Na comparação entre a abordagem híbrida e o DSE-

Ins completo, o fato da primeira ser mais lenta é uma aparente contradição, uma vez que o DSE-IC realiza uma única iteração de inserção sobre o espaço completo, enquanto o DSE-Ins realiza três iterações desse tipo. Entretanto, deve-se ter em conta que o DSE-IC parte de uma sequência inicial não vazia, o que acarreta mais posições a serem testadas para cada passo de otimização, explicando esse tempo de execução cerca de duas vezes mais elevado. O DSE-AG sobre o espaço completo levou, em média, 5509 segundos, enquanto que a exploração sobre o conjunto reduzido através de amostragem aleatória, o próprio AG, e dos algoritmos de inserção e de limpeza levou cerca de 422, 412, 97 e 19 segundos, ou seja, um ganho em torno de $13\times$, $13\times$, $57\times$ e $290\times$, respectivamente. Considerando-se as abordagens híbridas, os ganhos no tempo de execução do DSE sobre a abordagem DSE-AG completo foi de aproximadamente $20\times$ para o DSE-CI e $4\times$ para o DSE-IC.

Análise integrada das estratégias de DSE

No geral, os resultados indicam que as estratégias baseadas no conjunto completo conseguiram *speedups* melhores que aqueles obtidos pelos DSE sobre o espaço reduzido. Esse comportamento se deve, em boa parte, pela inexistência de algumas otimizações no espaço reduzido. A ausência ocorre por 2 razões: os passos de fato não estão presentes no conjunto de referência, sendo específicos da função alvo; ou, embora os passos estejam presentes nesse conjunto, foram utilizados em poucas funções de referência (uma ou duas) e as mesmas não fazem parte do grupo no qual se insere a função alvo após o processo de agrupamento. Enquanto a primeira está relacionada com a representatividade do conjunto adotado, a segunda está associada ao método de agrupamento, especialmente a forma de codificação puramente sintática. Em particular, como foi evidenciado nos experimentos, nosso método baseado em agrupamento não foi capaz de tratar adequadamente a função *dsp_mat_tr*. Essa função possui características que são difíceis de mapear, o que ficará evidenciado durante a investigação dos conjuntos de referência. Entretanto, a adoção das estratégias de DSE híbridas introduziu uma certa adaptabilidade à abordagem, permitindo superar os obstáculos encontrados e aproximar bastante o desempenho àquele obtido pelo DSE-AG completo, ao mesmo tempo que proporciona um ganho considerável no tempo de execução do DSE.

Em termos de tempo de execução do DSE, o uso de nossa abordagem propiciou uma redução considerável na quantidade de soluções avaliadas e, conseqüentemente, no tempo total da exploração. As explorações sobre o espaço reduzido foram mais rápidas, sendo que o algoritmo de limpeza foi o que apresentou os melhores tempos. Os tempos de exploração das abordagens híbridas também foram inferiores que aqueles gastos pelas abordagens que empregam AG e amostragem aleatória no espaço completo. Entretanto, o tempo médio de execução do DSE-IC é maior que aqueles relatados para as abordagens individuais sobre o espaço reduzido e o algoritmo de inserção no DSE completo. Em

contrapartida, o DSE-CI se mostrou bem competitivo, inclusive entre as abordagens que adotam exclusivamente o espaço reduzido, apresentando o terceiro melhor tempo médio de exploração (atrás apenas do DSE-Limp e DSE-Ins reduzido, respectivamente).

A análise do desempenho geral das diferentes estratégias de DSE pode ser feita sob uma ótica multiobjetivo. Um problema de otimização multiobjetivo (MOP) é caracterizado pela minimização e/ou maximização de duas ou mais funções objetivo, sendo que pode não haver uma melhor solução com respeito a todas essas funções. Entretanto, existe um conjunto de soluções que são superiores às demais dentro do espaço de busca. Este conjunto de soluções é conhecido como o Ótimo de Pareto ou soluções não dominadas (Srinivas e Deb, 1994). No processo de avaliação da eficácia das estratégias de DSE devemos levar em conta que existe um compromisso entre dois objetivos conflitantes: o tempo de execução despendido nessa exploração e o *speedup* médio retornado pela estratégia, considerando-se todas as funções alvo.

Em um MOP, pode não ser possível otimizar todos os objetivos ao mesmo tempo, pois ao otimizar um objetivo, é possível que outros sejam deteriorados. Nesses casos, pode não existir uma solução que possua, simultaneamente, valores ótimos para todos os objetivos no MOP em questão. Devido a esse conflito, utiliza-se a relação de dominância de Pareto.

Definição 1: Dominância de Pareto

Sejam duas soluções a e b do espaço de busca, diz-se que a domina b , representado por $a \succ b$, se e somente se: (i) a não é pior que b em todos os objetivos; (ii) a é estritamente melhor que b em pelo menos um objetivo.

A dominância de Pareto é uma relação de ordem parcial estrita e pode não ser possível estabelecer um ordenamento total das soluções, a partir da avaliação dos objetivos considerados. Isto significa que, dadas duas soluções a e b quaisquer, pode não ocorrer $a \succ b$ nem $b \prec a$. Assim, em um MOP é usual se buscar o conjunto de soluções não-dominadas P_0 :

Definição 2: Conjunto de soluções não-dominadas

Seja P um conjunto de soluções; $P_0 \subset P$ é o subconjunto composto por todas as soluções não-dominadas de P , ou seja, aquelas soluções em P para as quais não há outra solução em P que as domine.

Dessa forma, a otimização de Pareto pode ser vista como uma otimização onde a dominância global é o atributo desejado (Palermo et al, 2005).

A Figura 6.10 busca apresentar as diferentes estratégias de DSE investigadas, através da composição dos dois objetivos conflitantes em um único gráfico, permitindo a identificação das estratégias dominadas e do conjunto de não dominadas. Como os valores do tempo médio de execução do DSE (eixo y) apresentam uma grande variação entre as

abordagens, sendo o valor máximo de 5509 segundos e o mínimo de 19 segundos, eles foram exibidos na escala logarítmica. No gráfico é possível observar que a fronteira das soluções não-dominadas, contempla três abordagens, destacadas em círculos vermelhos: DSE-AG completo, DSE-CI e DSE-Limp. Todas as demais estratégias são dominadas por pelo menos um elemento desse conjunto. Por exemplo, embora a abordagem DSI-IC não seja dominada nem pela DSE-AG completo nem pelo DSE-Limp reduzido, ela é dominada pela estratégia DSE-CI, sendo então descartada do conjunto de soluções não-dominadas. Das três soluções que não são dominadas entre si, a abordagem DSE-AG completo, que é considerada nossa referência, é a que retorna o melhor *speedup* médio, mas a um alto custo computacional. A abordagem DSE-Limp reduzido é a mais rápida, porém apresenta um dos piores *speedups*. A abordagem DSE-CI, por outro lado, é a mais equilibrada delas, pois apresenta um *speedup* médio bastante próximo ao melhor possível (DSE-AG), a um custo de tempo bem abaixo das abordagens que usam o espaço completo.

Figura 6.10: Desempenho das estratégias de DSE avaliadas com um enfoque multi-objetivo.

Dessa forma, a partir desse ponto, consideraremos as estratégias DSE-CI e DSE-Limp nas análises subsequentes, por representarem as soluções no conjunto de Pareto que utilizam nossa abordagem.

6.4.2 Avaliação da eficácia do Método de Agrupamento

Duas investigações foram realizadas com o objetivo de verificar a importância do agrupamento realizado em nossa abordagem. Na primeira, o resultado obtido pelo método que indica a sequência de acordo com o agrupamento é confrontado com o *speedup* obtido pela aplicação da sequência de cada grupo para todas as funções alvo, independentemente do grupo na qual a função foi incluída (ou seja, sem o agrupamento). Na segunda, a utilização da composição das sequências do grupo é confrontada com a aplicação da melhor sequência do grupo, de forma isolada.

Análise dos grupos gerados pelo agrupamento

Na exploração baseada em agrupamento, as 46 funções usaram 12 grupos diferentes. Com o objetivo de avaliar a eficácia do método de seleção, nós repetimos o DSE usando cada espaço reduzido gerado pelos agrupamentos para explorar todas as 46 funções avaliadas. Esses espaços são formados pelos passos de compilação usados nas funções de referência presentes em cada grupo. A Tabela 6.5 apresenta as funções de referência agrupadas e o tamanho dos espaços gerados em cada grupo, bem como a quantidade de funções que utilizaram esses grupos durante a exploração. Apesar de diferentes, os grupos 2, 3, 11 e 12 geram espaços formados pelas mesmas otimizações, alternando apenas a ordem com a qual esses passos estão posicionados no subconjunto. O grupo 9, gerado durante a exploração da função *img_bounda*, resultou em um espaço que contém todas as otimizações do conjunto de referência (total de 20 passos). Ele é o subconjunto mais completo que pode ser gerado por qualquer combinação das funções de referência, portanto, será considerado nossa referência, já que não é possível gerar sequências melhores a partir do conjunto de referência adotado.

Na avaliação dos espaços reduzidos gerados a partir desses agrupamentos, foram investigadas as estratégias DSE-Limp e DSE-CI. A Figura 6.11 apresenta a média geométrica dos *speedups* alcançados pelas estratégias de DSE para as 46 funções avaliadas, considerando-se cada um dos 12 agrupamentos. Como pode ser observado na Figura 6.11a, nenhum outro grupo conseguiu superar o desempenho obtido a partir da exploração do nosso agrupamento pelo algoritmo de limpeza (1,33). Apesar de possuir todas as transformações do conjunto de referência, o grupo 9 atingiu o mesmo desempenho obtido pela nossa abordagem, indicando que o método de seleção foi capaz de agrupar adequadamente as otimizações potenciais, disponíveis no conjunto de referência. A grande maioria dos grupos resultaram em um *speedup* médio similar ou inferior ao obtido pelo DSE-Ins reduzido, nossa pior estratégia de exploração.

Na abordagem DSE-CI (Figura 6.11b), nosso agrupamento alcançou um *speedup* médio superior a todos os grupos avaliados (1,42), inclusive o grupo de referência (1,41). Os grupos com menor desempenho alcançaram um *speedup* médio ligeiramente superior a

Tabela 6.5: Agrupamentos utilizados nos experimentos.

Grupo	Funções de referência	Nº passos no grupo	Nº funções alvo
1	<i>autcor, bubble_sor, pop_cnt</i>	5	1
2	<i>autcor, pop_cnt, sobel, fdct, adpcm_code, adpcm_deco, bubble_sor</i>	17	7
3	<i>bubble_sor, sobel, fdct, adpcm_code, adpcm_deco</i>	17	10
4	<i>dotprod, dsp_minval, max, fibonacci</i>	10	1
5	<i>dsp_minval, bubble_sor</i>	7	1
6	<i>dsp_minval, dotprod, bubble_sor</i>	9	6
7	<i>dsp_minval, fibonacci, bubble_sor</i>	9	1
8	<i>dsp_minval, max, bubble_sor</i>	10	8
9*	<i>dsp_minval, sobel, fdct, adpcm_code, adpcm_deco, bubble_sor</i>	20	1
10	<i>fdct, sobel</i>	11	8
11	<i>sobel, fdct, adpcm_code, adpcm_deco</i>	17	1
12	<i>sobel, fdct, bubble_sor, adpcm_code, adpcm_deco</i>	17	1

* usado como agrupamento de referência

aqueles gerados pela abordagem baseada no conjunto reduzido e do DSE-Ins sobre o espaço completo. Em contrapartida, alguns grupos atingiram o valor de *speedup* médio do grupo de referência, o qual está bem próximo daquele obtido pelo DSE-AG na base completa (1,41 contra 1,43).

Dado que nas duas estratégias a utilização das otimizações de um único grupo não resultou em um *speedup* médio superior à nossa abordagem baseada em agrupamento, de um modo geral, os grupos indicados pelo método de seleção são adequados para a exploração do conjunto de funções. Além disso, os resultados obtidos evidenciam a capacidade de recuperação da abordagem DSE-CI, o qual foi capaz de melhorar significativamente o desempenho para todos os grupos avaliados, mesmo que não tenha conseguido atingir os melhores valores para todos os grupos.

Figura 6.11: Média geométrica dos *speedups* de acordo com o grupo usado pelo DSE.

Considerando cada função individualmente, o DSE-Limp não alcançou o *speedup* do

grupo de referência em 11 das 46 funções, enquanto que no DSE-CI, esse número caiu para apenas 4 funções. Essa diferença se deve aos casos que o DSE-Limp retirou, de maneira equivocada, uma otimização da sequência. Ou seja, todos os passos de otimização usados na "melhor" sequência gerada a partir dos agrupamentos estavam presentes no espaço reduzido. Isso ocorre porque o algoritmo de limpeza é um processo guloso, no qual as decisões são tomadas com base na situação de momento e não podem ser revistas ou desfeitas em um passo futuro. A decisão da remoção é afetada pela posição do passo na sequência atual, bem como pela própria composição da sequência (otimizações presentes). Dessa forma, a importância de uma otimização pode ser "mascarada" pela presença/aplicação de outras transformações presentes na sequência, as quais podem ser removidas futuramente. Portanto, isso é uma limitação do algoritmo de busca utilizado (estratégia de limpeza) e não do método de seleção. A estratégia híbrida DSE-CI executa uma iteração do algoritmo inserção a partir da sequência de saída do algoritmo de limpeza com objetivo de tornar a exploração menos dependente da ordem e composição do espaço reduzido. A Figura 6.12 apresenta os *speedups* médios alcançados pelo DSE-CI e pelo grupo de referência (grupo 9), assim como o maior valor encontrado para as 4 funções.

Como pode ser observado, nosso agrupamento resultou em *speedups* próximos aos maiores valores encontrados, sendo que a maior diferença foi de 0,04 na exploração da função *img_ycbcr4* e a menor foi de apenas 0,01 para a função *img_erode_*. Em relação ao grupo de referência, a abordagem híbrida com o processo de seleção só não atingiu o *speedup* para a função *img_ycbcr4*.

Figura 6.12: Melhor *speedup* alcançado pelo DSE-CI para as funções nas quais o método de seleção não atingiu o menor valor de latência dentre os grupos avaliados.

Exploração sem a composição das sequências

Além da exploração a partir dos conjuntos de otimizações gerado do agrupamento, também analisamos individualmente as sequências de otimização usadas por cada função

de referência. Esse experimento visa avaliar a necessidade da composição de diferentes sequências para o processo de DSE. Foram adotadas 4 abordagens distintas, que diferem entre si pela forma como a sequência é escolhida dentro do grupo e utilizada durante o DSE, como segue:

- **Sequência mais próxima:** a sequência de compilação da função de referência mais próxima (de acordo com a árvore filogenética) é aplicada diretamente na nova função.
- **Melhor sequência:** todas as sequências do grupo (cada uma isoladamente) são aplicadas diretamente sobre a nova função, adotando aquela que retorna o melhor desempenho (menor latência e maior *speedup*).
- **DSE baseado em limpeza a partir da melhor sequência:** cada sequência do grupo é explorada separadamente através da estratégia de DSE baseada no algoritmo de limpeza, sendo adotada aquela que obteve o melhor resultado.
- **DSE baseado no CI a partir da melhor sequência:** similar à abordagem anterior, mas que adota a estratégia híbrida CI (limpeza seguida da inserção) para explorar cada uma das sequências do grupo e também retorna o melhor resultado dentre elas. Nessa abordagem, a sequência explorada é a entrada do algoritmo de limpeza e o algoritmo de inserção utiliza todos os passos de otimização disponíveis no ambiente de compilação como espaço de busca (conjunto completo).

A Tabela 6.6 apresenta as médias geométricas dos *speedups* e dos tempos de execução gerados a partir da aplicação de cada uma das abordagens descritas sobre todas as funções alvo.

Tabela 6.6: Desempenho obtido a partir das abordagens baseadas nas sequências de otimização usadas pelas funções de referência do grupo da função alvo a ser compilada.

Abordagem	<i>Speedup</i> médio	Tempo médio de execução
Melhor sequência do grupo	1,31	8
Sequência mais próxima	1,18	2
DSE-Limp (melhor seq.)	1,33	26
DSE-CI (melhor seq.)	1,42	907

A abordagem baseada na sequência da função de referência mais próxima na árvore filogenética (vizinha imediata) obteve o menor tempo médio de execução (2 segundos), uma vez que envolve a avaliação de uma única sequência sobre o código. Entretanto, ela não conseguiu obter bons valores de *speedup* para as funções alvo, resultando em um comportamento médio bem inferior aos das outras abordagens. Isso ocorre porque a sequência das funções mais próximas nem sempre contém os passos de compilação necessários para a otimização de uma dada função alvo. A abordagem baseada na melhor

sequência do grupo também apresenta um tempo médio de execução muito baixo e, embora não tenha atingido o *speedup* médio das estratégias de DSE, chegou a valores bem próximos ao do DSE-Limp (1,31 contra 1,33). Entretanto, ela não elimina a necessidade do agrupamento, uma vez que trabalha sobre as funções de referencia dos grupos gerados pelo método de seleção. Sua utilização dispensa apenas a junção entre os passos de otimização das sequências presentes no grupo investigado (que contém a função alvo), e sua posterior exploração, o que justifica sua rapidez.

Em termos das estratégias de DSE, os resultados atingidos a partir da aplicação da melhor sequência do grupo foram muito parecidos com aqueles que obtivemos aplicando-se a composição das sequências do grupo, tanto para o DSE-CI, quanto para a exploração com o algoritmo de limpeza. Em contrapartida, a aplicação das estratégias de DSE sobre todas as sequências do grupo exige um tempo de exploração maior, dado que o algoritmo de busca é executado para cada sequência do grupo individualmente. No DSE-Limp, a exploração individual das sequências do grupo ocasionou um aumento de aproximadamente 39% no tempo de execução em relação à abordagem que combina os passos de otimização de todas as sequências. No DSE-CI esse aumento foi ainda maior, chegando ao percentual de 324%.

6.4.3 Análise das representações de entrada

A fim de ressaltar a relevância da representação DNA na identificação das estruturas importantes do código, outras formas de representação do código fonte foram usadas como entrada para o método de seleção e seus resultados foram comparados com aqueles obtidos com o uso do DNA. No escopo deste trabalho, realizamos agrupamentos a partir de 2 versões do código fonte: uma apenas com o código das funções (*kernel*) e outra contendo todo o programa usado nas simulações (*kernel* + função *main()* com a inicialização das variáveis). Também foram avaliados os agrupamentos a partir dos códigos *Assembly* gerados pelos compiladores ReflectC e GCC para o processador MicroBlaze. A utilização do GCC visa corroborar sobre a influência do compilador no código gerado, ou seja, o uso de compiladores diferentes pode resultar em códigos objeto distintos (inclusive em relação ao desempenho) para um mesmo processador.

Inicialmente, uma análise das árvores filogenéticas geradas para diferentes funções alvo foi realizada com o objetivo de avaliar o comportamento dos agrupamentos em função da forma de representação adotada para o código. Durante essa análise, foi possível verificar que o uso do código fonte não possibilita que nosso método identifique similaridades relativamente simples entre os códigos. Um exemplo disso pode ser observado nas árvores filogenéticas geradas para a função *autcor*—, as quais são apresentadas na Figura 6.13. Na árvore filogenética gerada a partir do código do *kernel* (Figura 6.13a), existem 4 nós intermediários entre as funções *autcor* e *autcor*— e, portanto, elas não

são colocadas no mesmo grupo. Nesse caso, o método de agrupamento não foi capaz de identificar a semelhança das funções, mesmo os seus códigos sendo quase idênticos. No árvore filogenética gerada a partir do código completo (Figura 6.13b), o caminho entre as funções é formado por 3 nós intermediários (situação limítrofe de agrupamento). Assim, apesar das funções ficarem no mesmo grupo, elas não estão próximas na árvore, ou seja, não são vizinhas. As árvores resultantes do *Assembly* gerado a partir do código fonte original (sem otimização), tanto pelo GCC (Figura 6.13c), quanto pelo ReflectC (Figura 6.13d), assim como aquela oriunda do DNA (Figura 6.13e), retrataram adequadamente a similaridade entre essas funções. Um comportamento similar também pode ser notado em relação às funções *max* e *dsp_minval* (maior e menor valor de um vetor), as quais só não são agrupadas a partir dos códigos fonte (*kernel* ou programa).

A fim de realizar uma análise do comportamento global, as diferentes fontes de entrada foram comparadas através do desempenho médio obtido por cada uma sobre as funções alvo. A Figura 6.14 mostra o gráfico com o *speedup* médio alcançado pelas estratégias de DSE baseadas no algoritmo de limpeza (DSE-Limpeza) e na abordagem híbrida CI (*Limpeza & Inserção*) para cada representação de entrada. O algoritmo de limpeza, por ser uma estratégia muito simples e altamente dependente do subconjunto de otimizações adotado, evidencia melhor a influência das formas de representação nos resultados atingidos. Por outro lado, os resultados obtidos com o DSE-CI demonstram a adaptabilidade desse método híbrido, uma vez que os desempenhos foram praticamente iguais para todas as formas de representação do código.

Em ambas as abordagens (DSE-Limpeza e DSE-CI), o uso do DNA e do *Assembly* gerado pelo ReflectC resultou nos melhores valores de *speedup* médio (1,33 e 1,42 respectivamente), indicando que essas formas de representação propiciam a identificação do mesmo padrão estrutural entre os códigos. Apesar dos resultados similares, os códigos DNA são muito menores que seus equivalentes em *Assembly*. Enquanto os arquivos com os códigos DNA de todas as funções avaliadas totalizam 5,8 *Kbytes*, os arquivos com os códigos *Assembly* correspondentes ocupam 86,6 *Kbytes*. Arquivos menores favorecem o processo de compressão durante o cálculo das matrizes de distância, considerando-se o tempo de execução necessário para realizá-lo. Além disso, ao contrário do *Assembly*, o uso do DNA torna o processo independente do compilador. Por utilizar outro tipo de compilador, o desempenho do DSE-Limpeza a partir do *Assembly* gerado pelo GCC (*speedup* de 1,30) só foi melhor que a exploração a partir do código fonte do *kernel*.

Dentre as explorações que utilizam o código fonte como entrada, o *speedup* médio obtido pelo DSE-Limpeza a partir do código completo (todo o programa) foi melhor do que aquele usando apenas o código do *kernel* (1,31 contra 1,24). No DSE-CI, eles atingiram o mesmo valor de *speedup* médio (1,41), demonstrando mais uma vez o poder de recuperação da abordagem. Ou seja, mesmo partindo de um subconjunto de otimizações menos representativo, a abordagem conseguiu gerar uma boa sequência de compilação,

Figura 6.13: Árvores filogenéticas geradas a partir das diferentes fontes de entrada para a função *autcor*—.

Figura 6.14: Desempenho das estratégias de DSE em função do tipo de representação do código usado como entrada para o método.

aproximando o desempenho ao do DSE-AG completo.

Além das diferentes fontes de entrada, também foi avaliada uma variação dos códigos *Assembly*, as quais foram geradas a partir dos códigos otimizados das funções. Na otimização foram empregadas as melhores sequências encontradas com o DSE-AG sobre o espaço completo para o ReflectC e o nível de otimização -O2 para o GCC. Apesar de não ser viável a adoção dessa estratégia pelo nosso método, uma vez que a sequência de otimização da função alvo ainda não é conhecida, esses testes foram realizados com o intuito de verificar a influência das otimizações nos agrupamentos. Os resultados indicaram que a aplicação das transformações sobre os códigos provoca mudanças nos agrupamentos que são prejudiciais ao desempenho do DSE. Em ambos os compiladores (ReflectC e GCC) houve uma redução de 1% no *speedup* médio obtido pelo DSE baseado no algoritmo de limpeza. De fato, as sequências de otimização são escolhidas de acordo com as especificidades de cada função. Portanto, passos de otimização diferentes podem ser aplicados sobre funções similares, diferenciando os códigos resultantes (otimizados). Além disso, funções inicialmente diferentes podem ser aproximadas pelas transformações em seus códigos. Esses comportamentos não são interessantes para nossa abordagem, uma vez que se deseja identificar os padrões de código que podem ser relacionados com as sequências de otimização a serem aplicadas.

6.4.4 Análise das regras de codificação do DNA

A representação em DNA consiste na conversão do código fonte da função em uma representação intermediária simplificada (com alguma perda de informação), através de um conjunto de regras de transformação. A escolha desse conjunto de regras pode influenciar no grau de precisão com o qual a representação do DNA reflete a sequência de *tokens* do código. Na nossa abordagem foram utilizadas as mesmas regras de transformação para a codificação do DNA usadas por Sanches et al (2011), as quais estão descritas na Tabela 6.7.

Tabela 6.7: Regras de transformação empregadas na codificação do DNA.

Símbolo no DNA	Elemento representado
H	<i>loops</i> do tipo <i>for</i>
I	Identificadores
A	Operadores aritméticos ('+' e '*')
Z	Constante zero
E	Atribuição ('=')
P	Operador '+='
Q	Operador de incremento ('++')
C	Operadores de comparação (ex: '==')
R	Operações de leitura da memória
W	Operações de escrita na memória

Essas regras são usadas para representar apenas o conteúdo dos laços (*loop*) existentes na função, desprezando todo o restante do código. Laços adjacentes são representados por cadeias de DNA distintas, cada qual ocupando uma linha no arquivo texto. Laços aninhados são apresentados na mesma linha. A justificativa dos autores é que os laços contêm a parte principal dos códigos avaliados e, portanto, é o que deve ser considerado para a distribuição das funções entre as unidades de processamento em *hardware*. Tal justificativa também pode ser estendida ao nosso contexto, uma vez que pretende-se otimizar funções dos domínios de processamento de sinais, imagens e vídeos, nos quais as partes críticas estão implementadas dentro de *loops*. Entretanto, para sustentar tal suposição, se faz necessária um estudo sobre o efeito da codificação de todo o código da função no resultado final do DSE. Além disso, apesar dos resultados obtidos nos experimentos indicarem a capacidade de agrupamento a partir do DNA, o conjunto de regras usado é muito simples. Neste experimento, investigamos a inclusão de novas regras de transformação com o objetivo de verificar se existe alguma informação/estrutura relevante que não está sendo tratada atualmente no DNA. Nesses novos experimentos, apenas a estratégia DSE-Limp foi avaliada, uma vez que ela se mostrou mais sensível a alterações do que a estratégia híbrida DSE-CI.

Para viabilizar tal avaliação, foi necessário replicar o sistema de codificação do DNA de

modo a permitir a inclusão das novas regras. Esse novo sistema foi implementado em JAVA dentro do ambiente de análise MANET, o qual é baseado no compilador fonte-para-fonte (*source-to-source*) CETUS. Durante esse processo, foram identificadas algumas conversões inadequadas na codificação do DNA original, as quais não foram implementadas no nosso sistema. São elas:

1.
 - O zero em hexadecimal não é representado no DNA.
 - Os identificadores criados a partir da diretiva de pré-compilação *#define* são representados como variáveis, sendo que, na nossa concepção, tais identificadores deveria ser tratados como constantes ou expressões, dependendo do seu conteúdo.
 - O operador de ponteiro '*' é representado como operador aritmético.
 - As chamadas de função são representadas como variáveis.
 - Os *arrays* presentes no lado esquerdo das expressões condicionais dos *ifs* são representados como operações de escrita ao invés de leitura.

As diferenças observadas ocasionaram pequenas alterações no *speedup* obtido para cinco funções (*dsp_firlms*, *dsp_mat_tr*, *img_bounda*, *img_corr_g* e *img/ycbcr4*). Entretanto, no geral, os *speedups* médios obtidos pelo DSE-Limp a partir das duas codificações de DNA foram similares (1,33).

Uma vez criado e validado o código de DNA base, o qual foi usado como referência para comparação, novas regras de transformação foram avaliadas para representar:

- **Estruturas *if*:** essa regra visa mapear desvios decorrentes das estruturas de seleção do tipo *if*.
- **Outros operadores aritméticos:** são introduzidos símbolos individuais para cada um dos operadores básicos aritméticos (soma, subtração, divisão e multiplicação), bem como para os operadores de decremento ('-') e operações simplificadas ('+=', '-=', '/=', '*=').
- **Tipos de dados (forma simplificada):** substitui o símbolo usado para identificadores por dois novos símbolos, um usado para representar variáveis inteiras (*int* e *char*) e o outro para representar variáveis de ponto flutuante (*float* e *double*).
- **Tipos de dados (forma detalhada):** é uma extensão da regra anterior, na qual é adotado um símbolo para cada tipo de dado tratado (*int*, *char*, *float* e *double*).
- **Tipos e modificadores de dados:** também é uma extensão da regra para tipos de dados simplificado, na qual é adicionado símbolos para representar modificadores

de dados (ex: *const*, *short*, *static*, etc.). Essa regra foi motivada por algumas observações a cerca dos códigos das funções e das suas sequências de otimização. Por exemplo, a adoção do modificador *short* pela função *dsp_minval* modifica completamente sua sequência de otimização em relação àquela usada pela função *max*.

- **Operadores de ponteiros:** insere dois novos símbolos para representar os operadores de manipulação de ponteiros '*' e '&'.
- **Chamada de funções:** adota um novo símbolo para representar as chamadas de funções.
- **Constantes:** estende o código DNA original, no qual somente o zero é representado, de modo que seu símbolo também seja usado para representar as demais constantes existentes no código.

As três últimas regras foram motivadas pelos problemas de representação identificados no código DNA original. Cada nova regra foi introduzida separadamente no conjunto de regras base, de modo a realizar a análise individual do seu efeito na exploração. Ao final, também foi avaliado um conjunto completo, formado pela combinação de todas as regras testadas.

O gráfico da Figura 6.15 ilustra o efeito da adoção de diferentes conjuntos de regras de transformação do DNA sobre o desempenho, em termos de *speedup* médio, da estratégia de DSE baseada no algoritmo de limpeza. As curvas do gráfico retratam as duas formas de aplicação do conjunto de regras sobre o código fonte da função. Na primeira, representada em preto, cada conjunto de regras é aplicado somente sobre as estruturas de laços do código. Na segunda, representada em vermelho, avaliamos o uso das regras de transformação sobre todo o código da função.

O desempenho do DSE ficou muito próximo em todos os cenários avaliados, apresentando uma pequena oscilação em torno de 1,31 e 1,33. As curvas geradas apresentam um contorno parecido, com algumas variações pontuais, indicando que o comportamento dos conjuntos de regras é estável em relação à forma como são aplicados (só laços ou todo o código), ou seja, seus efeitos são parecidos quando comparados com o desempenho do conjunto de regras original aplicado da mesma forma.

Em relação à forma da aplicação das regras, com exceção do conjunto completo, a curva gerada com o DNA só com os laços foi melhor que aquela gerada a partir do DNA de toda a função, indicando que a representação do restante do código dificulta a identificação dos elementos realmente relevantes para o agrupamento. Isso corrobora com a validade e manutenção da estratégia atual para a manipulação de funções cujos processamentos intensos estão nos corpos dos *loops*, como ocorre nos domínios de processamento de imagens e sinais.

Figura 6.15: *Speedup* médio obtido pelo DSE baseado no algoritmo de limpeza de acordo com o conjunto de regras de transformação e sua forma de aplicação durante a codificação do DNA.

Ao observar o desempenho entre os diferentes conjuntos de regras de transformação aplicadas apenas nos laços das funções, o ponto de máximo da curva foi obtido através da inclusão de símbolos para representar cada tipo de dados (representação detalhada) no conjunto original. O conjunto original também se mostrou robusto, apresentando um resultado similar ao melhor conjunto avaliado. A adoção de símbolos para representar os operadores de ponteiro e as chamadas de funções não alterou o desempenho do DSE em relação ao DNA original. Entretanto, durante a análise individual das poucas funções que possuem esses elementos, observamos que a inclusão do símbolo para operadores de ponteiro piorou o *speedup* das funções *dsp_dotp_s* (de 1,48 para 1,44) e *img_ycbcr4* (de 1,10 para 1,09), enquanto que a adoção da regra para a chamada de funções prejudicou o desempenho da função *dsp_fltoq1* (de 1,05 para 1,04). O uso dos demais conjuntos provocou uma pequena redução no *speedup* médio da exploração, sendo que o pior desempenho (ponto de mínimo da curva) foi obtido a partir do conjunto completo de regras de transformação. De fato, quanto mais informação for adicionada ao conjunto, mais próxima será a representação do DNA resultante do código fonte da função, o qual, como já foi verificado, não é uma boa fonte de entrada para o processo de agrupamento. Ao considerar o DNA gerado a partir de todo o código, o conjunto de regras original foi o que obteve o melhor *speedup* médio para o DSE, enquanto que a regra de representação de todas as constantes foi a que resultou no ponto de mínimo da curva.

No geral, o conjunto de regras original foi o que se mostrou mais estável quanto à forma de aplicação das regras sobre o código, retornando os melhores *speedups* (ou próximo deles). Vale destacar que as regras desse conjunto já vêm sendo estudadas há algum tempo (Sanches e Cardoso, 2008)(Sanches e Cardoso, 2010)(Sanches et al, 2011). Portanto, é esperado que elas sejam capazes de retratar de forma satisfatória as propriedades dos có-

digos fonte. Além disso, acreditamos ainda que para o contexto de compilação de códigos de *software* para processadores de propósito geral, o DNA adotado é suficiente para gerar os agrupamentos usados no processo de seleção do conjunto de passos de otimização. Entretanto, para síntese em *hardware*, várias das regras de codificação do DNA avaliadas nesse experimento são relevantes, tais como, tipos de dados e operações aritméticas. Assim, é de suma importância a adoção dessas regras, caso o método seja futuramente empregado na exploração de otimizações visando aceleradores em *hardware*.

6.4.5 Análise da matriz de dissimilaridades

Alguns experimentos foram executados com o objetivo de verificar a influência da forma de obtenção das matrizes de distâncias (dissimilaridades) na abordagem baseada em agrupamento. Durante os testes foram avaliados o uso de compressores distintos no cálculo das distâncias através da métrica NCD, bem como a adoção de outras métricas, não baseadas em compressão, para calcular a similaridade a partir dos códigos DNA das funções.

Tipos de compressores

A qualidade das matrizes de dissimilaridade geradas através da métrica NCD está relacionada com o tipo do compressor adotado no cálculo das distâncias (Cilibrasi e Vitanyi, 2005). Para assegurar uma boa matriz de distâncias, quatro diferentes bibliotecas de compressão de dados foram avaliadas em nossa abordagem: *bzlib* (utilizado em nossos experimentos), *ppmdx*, *lzmax* e *zlib*. Essas bibliotecas são integradas ao pacote CompLearn (Cilibrasi et al, 2008), o qual é usado pelo método de seleção para calcular as matrizes de distâncias.

Além do desempenho médio atingido pelo DSE aplicando-se as matrizes de distâncias geradas por cada um dos compressores, também foi avaliada a sua capacidade de compressão. Espera-se que quanto maior for a capacidade de compressão, mais precisa deve ser a aferição da dissimilaridade (distância) entre duas funções. Na análise dessa capacidade, o código DNA de todas as funções alvo e de referência foram colocados em um único arquivo texto, o qual foi então compactado através de cada biblioteca de compressão avaliada.

A taxa de compressão κ de um compressor x é então aferida em função do tamanho do arquivo original (T) e do arquivo compactado (T') a partir da Equação 6.2:

$$\kappa(x) = \left(1 - \frac{T'}{T}\right) \times 100 \quad (6.2)$$

A Tabela 6.8 apresenta os valores de *speedup* médio alcançados através das abordagens DSE-Limp e DSE-CI, bem como a taxa de compressão obtida para os diferentes algoritmos analisados.

Tabela 6.8: Desempenho dos compressores em relação aos *speedups* médios obtidos pelas abordagens DSE e à sua capacidade de compactação.

Compressor	DSE-Limp	DSE-CI	Percentual de compactação
<i>bzlib</i>	1,33	1,42	86,9%
<i>lzmax</i>	1,32	1,42	87,6%
<i>zlib</i>	1,32	1,42	87,6%
<i>ppmdx</i>	1,31	1,42	85,3%

Como podemos observar, os compressores apresentam uma capacidade de compactação similar. As maiores taxas de compactação foram atingidas através dos compressores *lzmax* e *zlib* (87,6% em ambos), enquanto o *ppmdx* resultou na menor taxa (85,3%). O *bzlib* obteve uma taxa de compressão intermediária (86,9%), mas que está mais próxima do valor atingido pelos compressores *lzmax* e *zlib* que do *ppmdx*.

Em relação ao desempenho das abordagens de exploração, o *bzlib* foi o compressor com o melhor *speedup* médio (1,33) para o DSE usando o algoritmo de limpeza, enquanto o *ppmdx* resultou novamente no pior valor (1,31). Na abordagem híbrida DSE-CI essa diferença foi anulada e todos os compressores resultaram no mesmo *speedup* médio (1,42).

De uma forma geral, os resultados mostram um equilíbrio entre os compressores. As bibliotecas de compressão *bzlib*, *lzmax* e *zlib* apresentam uma ligeira vantagem em relação ao *ppmdx*. Entretanto, acreditamos que qualquer um desses compressores pode ser adotado em nossa abordagem baseada em agrupamento.

Métricas de distância

Nosso método emprega a métrica da distância de compressão normalizada (NCD) para computar a matriz de distâncias entre funções. Essa métrica é inspirada na teoria da informação e calcula a dissimilaridade entre dois objetos a partir da compressão de seus dados. Nesse processo, a análise dos dados é feita de forma indireta, uma vez que a dissimilaridade é obtida em função do tamanho dos arquivos original e compactado (Equação 4.1).

Neste experimento, diferentes métricas de distância foram utilizadas para calcular as matrizes de distâncias a fim de avaliar sua influência no resultado final do DSE. Considerando que o código DNA é uma cadeia de símbolos, optamos por empregar distâncias específicas para lidar com cadeias de caracteres (*strings*), as quais analisam diretamente as sequências de símbolos, destacando suas diferenças. Mais especificamente, foram utilizadas algumas das distâncias fornecidas pelo pacote *stringdist* (van der Loo, 2014) implementado para a linguagem R. Esse pacote é uma versão estendida da função nativa *match* existente na linguagem, que inclui novos métodos para o cálculo de distâncias e a identificação aproximada de padrões entre *strings*. Dentre os vários tipos de distâncias dis-

ponibilizadas pelo pacote, nós utilizamos algumas métricas baseadas em edição e outras baseadas em granularidade (*q-gram*). São elas:

- **Distância de Levenshtein (LV):** contabiliza o menor número de operações de edição (deleções, inserções e substituições), que são necessárias para transformar uma *string* na outra.
- **Distância de Damerau-Levenshtein (DL):** contabiliza as mesmas operações da distância de Levenshtein, além de permitir a transposição entre símbolos adjacentes.
- **Distância baseada no alinhamento ótimo de *strings* (OSA - *Optimal String Alignment*):** uma versão restrita da distância de Damerau-Levenshtein, na qual cada *substring* só pode ser editada uma única vez. Por exemplo, um mesmo símbolo não pode sofrer duas transposições com o objetivo de avançá-lo na cadeia.
- **Distância da maior *substring* comum (LCS - *Longest Common Substring*):** contabiliza a quantidade mínima de símbolos que devem ser removido das duas *strings* até que as *substrings* resultantes sejam idênticas. Ela pode ser considerada uma caso especial de distância baseada em edição, na qual são permitidas apenas operações de inserção e deleção, desde que os símbolos remanescentes mantenham a mesma ordem usada nas *strings* originais.
- **Distância de granularidade-*q* (*q-gram*):** dado um vetor de granularidade-*q* para cada *string* (v_i e v_j), o qual é gerado a partir da contabilização do número de ocorrências de cada *substring* de *q* símbolos consecutivos existentes nas duas cadeias analisadas, essa distância corresponde à soma das diferenças absolutas entre esses vetores ($|v_i \times v_j|$).

Por exemplo, dadas duas *strings*: $S_1 = \text{"abcde"}$ e $S_2 = \text{"abdcde"}$. As possíveis *substrings* de 2 símbolos consecutivos ($q=2$) são:

$$\{\text{'ab'}, \text{'bc'}, \text{'cd'}, \text{'de'}, \text{'dc'}, \text{'bd'}\}$$

Seguindo essa ordem de *substrings*, os vetores de granularidade-2 para S_1 e S_2 são, respectivamente:

$$v_1 = \{1, 1, 1, 1, 0, 0\}$$

$$v_2 = \{1, 0, 1, 1, 1, 1\}$$

Logo,

$$|v_1 \times v_2| = 0+1+0+0+1+1 = 3$$

- **Distância de cosseno:** é o complemento da similaridade de cosseno entre os dois vetores de granularidade- q . A distância de cosseno (d_{cos}) é dada por:

$$d_{cos}(S_i, S_j) = 1 - \frac{|v_i \times v_j|}{\sqrt{|v_i|} \times \sqrt{|v_j|}} \quad (6.3)$$

sendo, $|v_x|$ a soma dos elementos do vetor v_x .

Por exemplo, considerando as *strings* e os vetores de granularidade do exemplo anterior, a distância de cosseno é:

$$d_{cos}(S_1, S_2) = 1 - \frac{3}{\sqrt{4 \times \sqrt{5}}} \cong 1 - \frac{3}{2 \times 2,236} \cong 1 - \frac{3}{4,472} \cong 1 - 0,671 \cong 0,329$$

- **Distância de Jaccard:** é o complemento do quociente da intersecção entre os vetores de granularidade- q (*substrings* compartilhadas) e a sua união (todas as *substrings* observadas). A distância de Jaccard ($d_{jaccard}$) é calculada por:

$$d_{jaccard}(S_i, S_j) = 1 - \frac{|v_i \cap v_j|}{|v_i \cup v_j|} \quad (6.4)$$

Considerando novamente o exemplo acima, temos:

$$|v_1 \cap v_2| = 1+0+1+1+0+0 = 3$$

$$|v_1 \cup v_2| = 1+1+1+1+1+1 = 6$$

Logo,

$$d_{jaccard}(S_1, S_2) = 1 - \frac{3}{6} = 1 - 0,5 = 0,5$$

As distâncias baseadas em granularidade foram avaliadas com três valores distintos para o parâmetro q ($q=\{1,2,3\}$). Para todas as métricas, $q=2$ resultou em um desempenho igual ou superior aos demais, portanto, iremos apresentar apenas os seus dados para efeito de comparação entre as métricas.

Além das distâncias do pacote *stringdist*, também foram implementadas duas novas métricas inspiradas nas distâncias por granularidade com $q=1$ (contabilização da ocorrência de símbolos), as quais não consideram a ordem dos símbolos no código DNA.

A primeira métrica, a qual denominamos distância do complemento da intersecção (DCI), é uma variação da distância de Jaccard, na qual é usado o complemento lógico da intersecção. Dessa forma, o cálculo da métrica DCI d_{dci} entre duas *strings* S_i e S_j é dado por:

$$d_{dci}(S_i, S_j) = \frac{|\sim (v_i \cap v_j)|}{|v_i \cup v_j|} = \frac{v_i \vee v_j}{v_i \vee v_j} \quad (6.5)$$

sendo, $\sim$ o operador lógico de negação (complemento lógico), $\vee$ o operador lógico "ou exclusivo" (*xor*), e $\vee$ o operador lógico "ou".

A segunda métrica, denominada distância das diferenças normalizadas ou DDN, computa a distância com base na soma dos valores normalizados das diferenças absolutas entre o número de ocorrências de cada símbolo do DNA. Considere uma *string* S_i escrita a partir de um alfabeto $\Sigma = \{a_1, a_2, a_3, \dots, a_n\}$. Para cada *string* S_i é contabilizada a quantidade de ocorrências do k -ésimo símbolo do alfabeto ($\tau(S_i, a_k)$). Então, a distância das diferenças normalizadas d_{ddn} entre duas *strings* S_i e S_j é calculada por:

$$d_{ddn}(S_i, S_j) = \frac{\sum_{k=1}^n |\tau(S_i, a_k) - \tau(S_j, a_k)|}{\sum_{k=1}^n \tau(S_i, a_k) + \tau(S_j, a_k)} \quad (6.6)$$

sendo, $|x|$ o valor absoluto de x .

A Figura 6.16 apresenta os *speedups* médios alcançados pelo algoritmo de limpeza e pela abordagem híbrida DSE-CI aplicados sobre os subconjuntos gerados por cada uma das distâncias avaliadas.

Figura 6.16: *Speedups* médios obtido pelo DSE baseado no algoritmo de limpeza de acordo com a métrica empregada no cálculo das matrizes de distâncias.

De uma forma geral, com exceção da distância de Jaccard, as métricas baseadas em granularidade (apresentadas em azul) apresentaram um desempenho melhor no DSE-Limp que aquelas baseadas em edição (apresentadas em verde). Enquanto as primeiras alcançaram valores de *speedup* próximos ao da NCD (destacada em vermelho), as métricas baseadas em edição tiveram um desempenho similar ao obtido pela abordagem DSE-Ins sobre o conjunto reduzido (pior resultado dentre as abordagens).

Analisando cada métrica individualmente, nenhuma conseguiu superar a NCD (em vermelho). A métrica da DDN conseguiu o mesmo *speedup* médio (1,33), e as distâncias

do cosseno (1,32), do complemento da intersecção (1,31) e de q -gram (1,31) também se aproximaram dos valores obtidos pela NCD. Essas quatro métricas são parecidas, dado que trabalham com a contabilização dos elementos de uma *string*. A diferença está no parâmetro q que define o tamanho dos elementos contabilizados (*substrings*) e na forma como essas métricas lidam de forma diferente com as quantidades contabilizada. Por exemplo, enquanto o cosseno trabalha com multiplicação e radiciação de pares de símbolos ($q=2$), a DDN trabalha com operações simples de soma e subtração sobre símbolos individuais. A diferença básica entre a DCI e a DDN é que enquanto a primeira apenas identifica se existe ou não um dado símbolo na *string*, a segunda contabiliza o número de ocorrências. Uma observação importante é o fato dessas métricas que não consideram a ordem dos símbolos no código, apenas sua existência e quantidade, resultarem em um desempenho similar ao de métricas que consideram tal ordem. Isso indica que a disposição dos elementos no código não influencia significativamente no agrupamento.

No DSE-CI, todas as métricas de distância resultaram em um desempenho similar, sendo possível sua distinção apenas na terceira casa decimal. Por exemplo, o pior *speedup* foi obtido a partir da distância do cosseno (1,4146), enquanto o melhor foi alcançado pela métrica DDN (1,4187). Todos esses valores obtidos estão próximos do *speedup* médio atingido pelo DSE-AG no espaço completo (1,4281), mostrando novamente o poder de recuperação da abordagem híbrida.

A partir dos resultados observados, concluímos que, as métricas NCD e DDN são as que melhor atendem as necessidades do nosso método, uma vez que apresenta o melhor desempenho dentre as distâncias avaliadas. Entretanto, ao analisar os *speedups* individuais de cada função, foi observado que outras métricas também conseguiram bons resultados pontuais, alguns atingindo valores próximos ou iguais aos obtidos pelo DSE-AG. Assim, acreditamos que o uso de uma abordagem que combine as árvores geradas a partir de um grupo de métricas de distância possa ser promissor e propomos essa investigação como trabalho futuro.

6.4.6 Análise do conjunto de referência

O método de seleção baseia-se em um conjunto de funções usadas como referência para realizar o agrupamento da nova função. Portanto, a eficiência desse agrupamento está diretamente ligada com a representatividade¹ do conjunto de referência. A simples inclusão de novas funções, sem uma avaliação prévia de sua relevância, não garante uma melhoria na qualidade da amostra, uma vez que funções similares podem não agregar novas possibilidades de agrupamento (novos passos de compilação ou grupos mais relevantes). Além disso, o aumento no tamanho do conjunto tem um impacto significativo no custo computa-

¹Neste contexto, representatividade esta associada à razão entre o número de passos de otimização usados pela função em relação à quantidade de passos de otimização usados pelas funções vizinhas.

cional do processo de seleção. Por exemplo, cada função de referência é um nó folha da árvore filogenética e o aumento da quantidade de folhas provoca um crescimento fatorial no número de topologias possíveis (da Silva, 2007).

Para obter uma boa amostragem do espaço de funções, o conjunto de referência deve ser formado por pontos bem distribuídos do espaço de funções e cada ponto deve ser representativo dentro de sua região. Dependendo do tamanho desse espaço e da quantidade que será utilizada como referência, a busca exaustiva pelo melhor conjunto não é factível sem alguma restrição do domínio (ex: apenas funções de processamento de imagem). Outra alternativa é adotar critérios para suportar uma escolha sistemática das funções que devem compor o conjunto de referência.

A fim de verificar o impacto das funções utilizadas no conjunto de referência no *speedup* médio alcançado pelo DSE, adotamos diferentes conjuntos na exploração através do algoritmo de limpeza. Essa estratégia foi escolhida por ser o mais simples e rápido, propiciando a análise de uma maior quantidade de conjuntos promissores. Além disso, ele também é o mais afetado pela qualidade do conjunto escolhido, dado que seu desempenho é fortemente influenciado pela configuração do espaço reduzido gerado a partir do agrupamento. Neste estudo, foi mantida a quantidade de funções do conjunto original (11 funções) para simplificar o espaço de busca e facilitar a comparação dos resultados. Os novos conjuntos de referência foram gerados de acordo com seis diferentes estratégias para a seleção das funções, como segue:

- **Amostragem aleatória:** consiste na seleção aleatória de 11 funções dentre as 57 existentes. Como não existe qualquer direcionamento nessa escolha, acreditamos que os conjuntos gerados não possuem a representatividade necessária para um bom agrupamento. Entretanto, tal estratégia é interessante para avaliar o desempenho das demais. Neste experimento foram avaliados 30 conjuntos gerados de forma totalmente aleatória.
- **Seleção da melhor cobertura global:** essa estratégia visa selecionar funções de modo a garantir que todos os 33 passos de otimização existentes nas sequências "ótimas" obtidas pelo Algoritmo Genético para as 57 funções (alvo e referência) estejam presentes no conjunto de referência. A ideia é tentar evitar, ou pelo menos minimizar, que a quantidade de passos coberta pelo conjunto de referência limite o desempenho que pode ser alcançado pela abordagem, como ocorre com o conjunto atual que cobre apenas 20 passos dos 33 necessários. Inicialmente, foram selecionadas as funções que empregam otimizações com a menor quantidade de ocorrências entre as sequências. Essas funções são então combinadas de modo a garantir a presença de todos os passos no conjunto de referência. Nessa etapa, também foi utilizada a árvore filogenética gerada a partir do código DNA das funções para selecionar entre aquelas que possuem a mesma quantidade de otimizações relevantes,

escolhendo as funções que estavam em regiões da árvore sem ou com pouca representação. Essa estratégia visa garantir uma melhor distribuição espacial do conjunto na árvore. Como a cobertura total dos 33 passos de otimização pode ser obtida a partir da combinação de um número menor de funções, o conjunto de referência é complementado com funções que, apesar de não agregarem novas otimizações, fornecem uma redundância entre os passos mais raros (com menor ocorrência no conjunto), e garantem uma boa distribuição espacial na árvore. Nesse processo, 21 funções foram utilizadas para gerar 22 conjuntos de referência potenciais.

- **Seleção da melhor cobertura de um subconjunto:** essa estratégia mistura as duas abordagens anteriores. Ela consiste na seleção do conjunto de referência a partir do critério de cobertura sobre um subconjunto de 26 funções escolhidas aleatoriamente. Como não há garantia que todos os passos de otimização estarão presentes no subconjunto selecionado, existe uma forte tendência que a cobertura do conjunto de referência seja parcial. Essa estratégia visa reproduzir o cenário que seria encontrado pelo método quando aplicado em situações reais, ou seja, em um ambiente não controlado, onde dificilmente o conjunto de referência cobrirá todas as regiões possíveis dos espaços de otimizações e de funções. Nesses casos, o conjunto de referência seria obtido a partir de um subconjunto de funções já conhecidas. Nesse experimento foram avaliados 30 conjuntos, cada qual selecionado a partir de um subconjunto diferente gerado de forma totalmente aleatória.
- **Seleção da melhor cobertura por grupo:** essa estratégia parte do agrupamento das 57 funções manipuladas em nossos experimentos em 11 grupos distintos. Esse agrupamento foi feito usando a árvore filogenética gerada a partir do DNA das funções e utilizou uma versão do algoritmo *Fast Newman* (Newman, 2010) disponibilizada no pacote MIToobox da Universidade de Manchester (of Manchester, 2014). Esse algoritmo costuma ser empregado no agrupamento de redes complexas e foi adotado por permitir o agrupamento a partir de árvores filogenéticas (trabalha com matriz de adjacências) e a escolha do nível de agrupamento desejado (quantidade de grupos). Os 11 grupos gerados pelo algoritmo *Fast Newman* são apresentados na Tabela 6.9.

Para cada grupo, é identificada uma função que, quando combinada com as selecionadas dos outros grupos, resulte no maior subconjunto de passos de otimização distintos, ou seja, são selecionadas as funções de cada grupo que propiciem a melhor cobertura do conjunto de otimizações usadas por todas as funções. Ao final dessa análise, foram identificados 128 arranjos diferentes, os quais conseguem representar 29 das 33 otimizações utilizadas.

- **Seleção das maiores sequências globais:** essa estratégia de seleção prioriza as

Tabela 6.9: Agrupamento de todas as funções manipuladas em 11 grupos distintos pelo algoritmo *Fast Newman*.

Grupo	Funções
1	<i>dsp_maxval, dsp_maxidx, dsp_minval, goraud—</i>
2	<i>dsp_blk_e0, dsp_blk_e1, dsp_blk_e2, dsp_mul32, img_perime, img_yc_de0, img_yc_de1</i>
3	<i>img_dilate, img_erode_, img_median, img_sobel, sobel</i>
4	<i>dsp_blk_mo, dsp_neg32, dsp_q15tof, dsp_w_vec, img_dilate, img_pix_ex</i>
5	<i>autcor, bubble_sor, dotprod, fibonacci, max, pop_cnt</i>
6	<i>dsp_fltoq1, dsp_mat_tr, img_pix_sa</i>
7	<i>adpcm_code, adpcm_deco, dsp_fir_cp, img_conv_3, img_ybcr4</i>
8	<i>dsp_bexp_cp, dsp_dotpro, dsp_dotp_s, dsp_firlms, dsp_vecsum, img_sad_8x, img_sad_16</i>
9	<i>autcor—, dsp_autoco, dsp_minerr, img_bounda, img_corr_g</i>
10	<i>fdct, fdct—, img_fdct_8, img_idct_8</i>
11	<i>dsp_mat_mu, img_mad_8x, img_mad_16, img_quanti, img_wave_h, img_wave_v</i>

funções com as maiores sequências de otimização. Tal ideia parte do suposto que sequências maiores possuem mais transformações e, portanto, aumentam a probabilidade de geração de uma boa sequência de otimização mesmo a partir de agrupamentos com poucas funções. Inicialmente foram selecionadas as 11 funções com as maiores sequências, cujo tamanho variam de 8 a 16 passos de otimização. A fim de gerar um número maior de alternativas, foram analisadas as semelhanças entre as sequências e os códigos de DNA (proximidade na árvore filogenética) das funções e identificadas aquelas que supostamente são similares e, portanto, não agregam informação adicional para o agrupamento. A partir dessa análise, cinco cenários foram considerados: (i) *fdct*, *fdct—* possuem o mesmo código, mudando apenas o parâmetro de configuração, e a mesma sequência de otimização; (ii) *img_dilate* e *img_erode_* são vizinhas na árvore e adotam sequências muito parecidas (*img_dilate* adota 2 passos diferentes, enquanto *img_erode_* apenas um passo); (iii) *img_idct_8* e *img_fdct_8* são vizinhas na árvore e utilizam sequências similares, tanto no tamanho (13 passos de otimização), quanto na sua composição (diferem em apenas 3 passos), entretanto, *img_fdct_8* possui uma otimização que não está representada em outras funções selecionadas; (iv) *adpcm_deco* e *adpcm_code* são similares, sendo a sequência da primeira menor que a da segunda; e (v) *dsp_bexp_c* e *dsp_fir_cp* adotam sequências muito parecidas, que envolvem varias repetições do *loopguard* e *loopcanon* e, apesar da *dsp_fir_cp* usar 2 passos distintos a mais (os quais também estão nas sequências de outras funções), a distribuição da *dsp_bexp_c* na árvore a torna uma função mais interessante. Essas considerações possibilitaram a criação de outros 31 conjuntos, totalizando os 32

avaliados nesta estratégia.

- **Seleção das maiores sequências por grupo:** a partir do agrupamento das 57 funções em 11 grupos, foram identificadas as funções com as maiores sequências de cada grupo. Nesse processo, foram utilizadas 2 abordagens: com e sem repetição de passos de otimização. A segunda abordagem possibilita a melhor cobertura local por grupo, ou seja, a sequência com a melhor cobertura dentro de cada grupo sem considerar as escolhas feitas nos outros grupos. As funções selecionadas pelas duas abordagens foram então combinadas entre si, mantendo a representatividade de cada grupo, formando os 8 conjuntos de referência desta estratégia. A relação das funções adotadas em cada grupo é apresentada na Tabela 6.10.

Tabela 6.10: Funções com as maiores sequências de cada um dos 11 grupos gerados.

Grupo	Função adotada
1	<i>goraud</i> —
2	<i>dsp_blk_e1</i> ou <i>img_perime</i>
3	<i>img_dilate</i>
4	<i>dsp_w_vec</i>
5	<i>max</i>
6	<i>dsp_mat_tr</i>
7	<i>dsp_fir_cp</i> (considerando os passos repetidos) ou <i>adpcm_code</i> (somente passos distintos)
8	<i>dsp_bexp_cp</i> (com e sem repetição)
9	<i>img_bounda</i>
10	<i>img_fdct_8</i> ou <i>img_idct_8</i>
11	<i>dsp_mat_mu</i>

Ao total foram avaliados 258 conjuntos diferentes. O objetivo é investigar o impacto desses critérios de seleção nos resultados alcançados por nossa abordagem. Cada conjunto gerado foi individualmente aplicado na exploração das funções remanescentes através do algoritmo de limpeza. Para viabilizar a comparação entre os diferentes conjuntos (inclusive o conjunto original), a média geométrica foi calculada a partir dos *speedups* individuais alcançado para todas as funções (de referência e alvo). Dessa forma, as explorações com o conjunto de referência original (Martins et al, 2014b)(Martins et al, 2014a) atingiram um *speedup* médio de 1,33 para o DSE-Limp, 1,42 para o DSE-CI e 1,43 para o DSE-AG completo. Além disso, como em cada estratégia de seleção foram geradas diferentes quantidades de conjuntos de referência potenciais, a comparação entre elas foi realizada através de algumas métricas comumente usadas em estatística: média aritmética, desvio padrão, média geométrica, mediana, e melhor e pior caso (maior e menor valores dos *speedups* médios atingidos pelos seus conjuntos).

A Tabela 6.11 apresenta os valores das métricas estatísticas obtidos para cada estratégia usada na seleção das funções de referência, sendo que seus comportamentos são

evidenciados através das curvas no gráfico da Figura 6.17. Neste gráfico também são apresentadas duas linhas referentes aos *speedups* médios alcançados pelo conjunto de referência original usando o DSE-Limp e o DSE-AG completo, as quais delimitam a faixa considerada satisfatória para uma boa estratégia de seleção.

Tabela 6.11: Métricas estatísticas geradas a partir dos conjuntos avaliados em cada estratégia de seleção das funções de referência.

Estratégia de Seleção	Melhor caso	Média aritmética	Desvio padrão	Mediana	Média geométrica	Pior caso
Cobertura por grupo	1,40	1,38	0,010	1,37	1,38	1,35
Cobertura global	1,40	1,36	0,027	1,36	1,36	1,29
Cobertura do subconjunto	1,38	1,33	0,031	1,32	1,33	1,27
Amostragem aleatória	1,38	1,33	0,031	1,33	1,33	1,25
> sequências globais	1,35	1,32	0,015	1,32	1,32	1,28
> sequências por grupo	1,33	1,33	0,004	1,32	1,33	1,32

Como pode ser observado, não houve diferença entre os valores das médias aritmética e geométrica obtidas em cada estratégia de seleção, indicando que a distribuição dos *speedups* médios dos conjuntos seguem a tendência central do grupo avaliado na estratégia. Portanto, não faremos distinção entre elas durante a comparação das estratégias.

Figura 6.17: Comportamento dos conjuntos avaliados de acordo com o critério utilizado para a seleção das funções de referência.

O uso do critério de melhor cobertura a partir do agrupamento e sobre todas as funções resultou nos maiores *speedups* médios entre todos os conjuntos avaliados (cerca de 1,40 em ambos), bem como nas maiores médias (1,38 para a cobertura por grupo e 1,36 para a cobertura global) e medianas (1,37 e 1,36, respectivamente). Entretanto, o pior conjunto da cobertura global obteve um *speedup* médio de 1,29, o qual é inferior aos valores obtidos pelo conjunto de referência original (1,33) e pelos piores conjuntos das abordagens

baseadas nos grupos (1,35 para a melhor cobertura e 1,32 para as maiores sequências).

Os critérios de seleção baseados nas maiores sequências, tanto geral quanto por grupo, foram os que apresentaram os menores valores de *speedup* médio entre os melhores casos de cada estratégia (1,33 no critério que considera os grupos e 1,35 no global). Entretanto, conforme mostrado pelas curvas do gráfico, suas médias e medianas ficaram entre os valores obtidos pelas abordagens que usam aleatoriedade, superando-os nos piores casos. Dentre os piores casos, o conjunto gerado pela adoção das maiores sequências por grupo só foi pior que aquele obtido pela melhor cobertura por grupo (1,32 contra 1,35).

As estratégias baseadas em aleatoriedade, ou seja, melhor cobertura do subconjunto aleatório e amostragem aleatória, possuem curvas com um comportamento muito parecido. O maior *speedup* médio alcançado pelos conjuntos de cada uma dessas estratégias foi de 1,38, ficando abaixo dos melhores casos das outras abordagens baseadas em cobertura e acima das estratégias que usam as funções com as maiores sequências. Os valores de suas médias e medianas também estão muito próximos entre si. Enquanto a amostragem aleatória resultou na terceira melhor média, 1,33 para as médias e a mediana, o uso da melhor cobertura do subconjunto aleatório também obteve 1,33 de média, mas 1,32 de mediana (a pior entre os critérios avaliados). Em contrapartida, ao analisarmos o pior caso, as estratégias aleatórias tiveram os menores *speedups* médios, sendo a abordagem baseada na cobertura do subconjunto um pouco melhor (1,27 contra 1,25).

O critério de seleção a partir do agrupamento resultou no maior valor de *speedup* dentre todos os conjuntos avaliados (1,40 para a abordagem baseada na melhor cobertura). Além disso, também obtiveram a menor oscilação entre os desempenhos de seus conjuntos, tanto para melhor cobertura, quanto para a maior sequência, indicando um maior equilíbrio entre os conjuntos gerados e, conseqüentemente, uma melhor estabilidade nos *speedups* médios atingidos. Entre as estratégias que empregam esse critério, a adoção das maiores sequência foi a mais estável, apresentando um desvio padrão de 0,004, enquanto que a abordagem baseada na cobertura teve um desvio padrão de 0,010. Em contrapartida, o critério de melhor cobertura alcançou os melhores valores em todas as métricas de comparação: melhor caso (1,40 contra 1,33), médias (1,38 contra 1,33), mediana (1,37 contra 1,32) e o pior caso (1,35 contra 1,32).

Considerando os valores apresentados e os comportamentos das curvas no gráfico, concluímos que as estratégias que usam as maiores sequências ou algum componente aleatório (amostragem aleatória e melhor cobertura sobre um subconjunto aleatório) não são adequados para a seleção das funções de referência, dado que apresentam, na média, um desempenho menor ou igual ao obtido pelo conjunto de referência original. Nos critérios baseados nas maiores sequências, assim como nas abordagens aleatórias, apenas os melhores casos foram superiores ao *speedup* médio do conjunto original. Apesar de atingir um desempenho superior ao do conjunto de referência original tanto no melhor caso, quanto nas médias e mediana, a estratégia que usa a melhor cobertura global (con-

siderando todas as funções) apresentou uma grande variação entre os valores máximo e mínimo de *speedup*, resultando, no pior caso, em um valor muito abaixo do conjunto original. A estratégia de seleção das funções por grupo que garante a melhor cobertura dos passos de otimização foi a única que atingiu um desempenho superior ao do conjunto de referência original em 100% dos conjuntos avaliados. Portanto, consideramos que essa abordagem é a mais adequada para a escolha sistemática de bons conjuntos de referência.

6.4.7 Desdobramento de laços (*loop unrolling*)

Neste experimento, investigamos como nossa abordagem se comporta na exploração de sequências a partir de um conjunto que contenha passos de otimização que envolvam a configuração de parâmetros. Para isso, uma nova transformação, denominada desdobramento de laços (*loop unrolling*), foi introduzida ao conjunto de passo de otimização utilizado. Ela é responsável pela redução no número de iterações de uma estrutura de repetição através do seu desdobramento. Isso implica na replicação do corpo do *loop* uma quantidade N de vezes, sendo N o fator de desdobramento aplicado pela transformação. Portanto, junto com o *unrolling*, também devem ser fornecidas outras informações, como a identificação do *loop* no qual será aplicada a transformação e o seu fator de desdobramento. A configuração adequada desses parâmetros aumenta a complexidade da busca durante o DSE.

O *loop unrolling* é um passo de otimização que influencia significativamente na melhoria do *speedup* de algumas aplicações. Sua exploração é particularmente relevante em sistemas com cache, nos quais a efetividade do desdobramento é limitada pelo tamanho da cache de modo a evitar *stalls*. Entretanto, mesmo em arquiteturas sem cache, nas quais o desdobramento costuma resultar em melhores *speedups*, existem situações em que a exploração do *loop unrolling* se faz necessária. Por exemplo, nas abordagens *multicore* usadas atualmente, os *cores* são geralmente simples e não possuem cache, mas o fator do desdobramento é influenciado pelo paralelismo.

Neste contexto, avaliamos duas abordagens de exploração baseadas no DSE-AG sobre o conjunto completo de passos de otimização, agora formado por 50 passos de otimização.

A primeira, chamada aqui de *AG de etapa simples*, é uma extensão da abordagem original, na qual os fatores de desdobramento também são escolhidos aleatoriamente a partir de um conjunto de opções pré-definido. Neste trabalho, empregamos 3 fatores de desdobramento (1, 2 ou 4). Apesar do fator 1 não aplicar o desdobramento de fato, ele realiza uma série de transformações de preparação do código que podem melhorar o desempenho. Na geração dos filhos, um operador de *crossover* com um único ponto de corte comum é aplicado sobre os cromossomos dos pais e realizada a troca genética. O operador de mutação consiste na troca de um fator de desdobramento existente no cromossomo

com outro contido no espaço. Ambos operadores genéticos são aplicados com uma certa probabilidade, a qual é selecionada de forma aleatória.

Na segunda abordagem, chamada de *AG de duas etapas*, aplicamos a técnica de subdivisão do problema (descrita na Seção 3.3) para reduzir a complexidade da exploração. Assim, nossa exploração é realizada em duas etapas: (i) escolha do melhor fator de desdobramento, e (ii) sequenciamento dos passos de otimização. Na primeira etapa, explora-se todas as possíveis configurações de fatores de desdobramento (busca exaustiva) para cada laço (*loop*) do código, escolhendo aquela que atinge o melhor *speedup* para a função quando somente o desdobramento é aplicado. Essa configuração será então utilizada durante todo o processo de exploração da sequência de otimização. Se não existir nenhuma configuração de fatores capaz de melhorar o desempenho do código original (sem otimização), o *loop unrolling* é removido do espaço de otimizações. Os dois casos (escolha prévia do fator de desdobramento ou a remoção do passo) propiciam que a segunda etapa da exploração seja reduzida a um espaço de busca sem parâmetros de configuração adicionais, ou seja, como foi realizado na abordagem original descrita na Seção 6.4.1, sem a necessidade de ajustar algum parâmetro do passo de otimização (o fator de desdobramento de cada *loop*). A Tabela 6.12 apresenta os valores de *speedup* médio e tempo médio de execução para as abordagens de DSE-AG sobre o espaço completo de otimizações.

Tabela 6.12: Desempenho das duas abordagens de DSE-AG usando um espaço completo com *loop unrolling*.

Estratégia DSE	Speedup médio	Tempo médio de execução
Sem <i>loop unrolling</i>	1,43	6168
AG de etapa simples (com <i>loop unrolling</i>)	1,61	11292
AG de duas etapas (com <i>loop unrolling</i>)	1,63	10138

No geral, as abordagens são bastante similares, sendo que a exploração em duas etapas (fatores e sequenciamento) resultou no melhor valor de *speedup* médio e também no menor tempo de execução. A inclusão do *unrolling* na exploração propiciou um aumento no *speedup* médio de 0,18 e 0,20 para a abordagem de etapa simples e a de duas etapas, respectivamente, o que resulta em um ganho em torno de 13% nas duas abordagens. Por outro lado, o tempo médio da exploração aumentou consideravelmente, cerca de 83% na abordagem de etapa simples e 64% na abordagem de duas etapas.

Analisando-se as funções individualmente, observamos algumas diferenças tanto no *speedup*, quanto nos fatores adotados. Em sete funções, a abordagem simples resultou em *speedups* melhores que os da exploração em duas etapas que, por sua vez, obteve os melhores *speedups* em nove funções. Parte dessas diferenças se deve pelo comportamento estocástico do AG (ex: *dsp_fir_cp*). Outras quatro situações foram provocadas pelo uso de diferentes fatores de desdobramento entre as abordagens (*adpcm_deco*, *fdct*—,

img_bound e *img_corr_g*), sendo que sempre a melhor latência foi obtida a partir do maior fator de desdobramento. Ao analisar esses casos, foi possível observar que a abordagem em 2 etapas não foi capaz de identificar o melhor fator de desdobramento porque ele só é viável após a aplicação de outros passos de compilação. Durante a 1ª etapa (exploração do desdobramento), o uso desses fatores resultaram em soluções inválidas ou em códigos com maior latência. A adoção do *loop unrolling* tanto na função *autcor*— quanto na função *autcor* é coerente, dado que seus códigos são muito similares, e se seu uso é indicado para uma, também deveria ser interessante para a outra. Entretanto, o mesmo não ocorreu com *fdct* e *fdct*—, as quais também tem códigos parecidos, mas somente para uma delas o AG indicou o uso do *loop unrolling*. Esse comportamento ocorreu nas duas abordagens avaliadas. Portanto, foi realizado um teste cruzado, no qual as sequências são aplicadas de forma invertida (sequência de uma aplicada na outra e vice-versa), a fim de verificar se os resultados estão corretos ou foi o AG que não conseguiu identificar a necessidade da otimização na sequência do *fdct*. Além do teste cruzado, também foi avaliada a sequência da *fdct* com a inclusão do *loop unrolling* no início, meio e fim da mesma. As 5 sequências avaliadas são apresentadas a seguir e os resultados obtidos estão na Tabela 6.13:

- *fdct*: *loopprev*, *lrrename*, *loopive*, *loopstrength*, *vprop*, *loopguard*, *dismemun*, *cse*, *loopbcount*, *strength*
- *fdct*—: *dismemun*, *loopguard*, *strength*, *loopive*, *vprop*, *loopstrength*, *loopprev*, *unroll*, *constprop*, *cse*
- *fdct* com *unrolling* no início: *unroll*, *loopprev*, *lrrename*, *loopive*, *loopstrength*, *vprop*, *loopguard*, *dismemun*, *cse*, *loopbcount*, *strength*
- *fdct* com *unrolling* no meio: *loopprev*, *lrrename*, *loopive*, *loopstrength*, *vprop*, *unroll*, *loopguard*, *dismemun*, *cse*, *loopbcount*, *strength*
- *fdct* com *unrolling* no fim: *loopprev*, *lrrename*, *loopive*, *loopstrength*, *vprop*, *loopguard*, *dismemun*, *cse*, *loopbcount*, *strength*, *unroll*

Todas as sequências foram avaliadas com a configuração resultado pelo AG de duas etapas para o *fdct*—, ou seja, a aplicação de um fator de 4 somente para o primeiro *loop*. Os resultados desses testes são apresentadas na Tabela 6.13.

Como pode ser observado, ao usar as sequências invertidas, ocorre um aumento, mesmo que pequeno, nas latências (*fdct* aumentou de 30985 para 32089 e *fdct*— de 12802 para 12810). Analisando o código fonte das funções, verificamos que o número de transformadas de cosseno afeta no comportamento da função, de modo que se igualarmos seus valores, os desempenhos a partir das sequências ficam idênticos. O uso da sequência da *fdct* com o *loop unrolling* no meio ou no final não melhora a latência obtida sem o passo. Entretanto, ao incluí-lo no início da sequência da *fdct*, foram alcançadas

Tabela 6.13: Latência obtida a partir da aplicação de sequências com e sem *loop unrolling* sobre as funções *fdct* e *fdct——*.

Sequência	<i>fdct</i>	<i>fdct——</i>
<i>fdct</i>	30985	12810
<i>fdct——</i>	32089	12802
<i>fdct</i> com <i>unrolling</i> no início	30871	12658
<i>fdct</i> com <i>unrolling</i> no meio	31332	12935
<i>fdct</i> com <i>unrolling</i> no fim	31255	12903

as melhores latências para as duas funções. Isto mostra que a posição do *loop unrolling* influencia muito no resultado, assim como o AG não está adaptado para manipular esse tipo de especificidade.

Outro ponto evidenciado durante os experimentos é o fato do uso do *loop unrolling* provocar um grande número de soluções inválidas, ou seja, o compilador ReflectC não consegue aplicar a sequência sobre o código. Quando isso ocorre, um valor elevado é atribuído para a latência. Nas estratégias de DSE baseadas em população (AG e amostragem aleatória), os algoritmos são capazes de escapar das regiões inválidas, usando o valor elevado de latência para tratar tais soluções e destacar soluções mais promissoras. Entretanto, para as abordagens baseadas em algoritmos gulosos, como os algoritmos de limpeza e de inserção, tais ocorrências são prejudiciais ao bom funcionamento do método de busca. Por exemplo, no DSE-Limp, se a sequência com *unrolling* retornar um resultado inválido, dependendo da posição do passo na sequência, o algoritmo pode ser induzido a retirar passos relevantes ao problema, ou até mesmo toda a sequência de otimização. Dessa forma, é necessária uma adequação dos métodos determinísticos de busca baseados em limpeza (híbridos e simples), para viabilizar a realização de experimentos com o *loop unrolling* na abordagem baseada em agrupamento. Uma possível solução vislumbrada consiste na adoção de uma variação da abordagem em duas etapas adotada para o AG. Nessa nova abordagem, a sequência inicial é manipulada de modo a posicionar o *loop unrolling* no início. Isso possibilita a identificação da validade do *loop unrolling* antes de manipular as demais otimizações, uma vez que o passo é removido caso a sua aplicação resulte em uma sequência inválida. A Tabela 6.14 compara a melhoria de desempenho alcançada pelas três estratégias de DSE não dominadas (DSA-AG completo, DSE-Limp e DSE-CI) a partir da adoção do *loop unrolling*. Ela apresenta as médias geométricas dos *speedups* obtidas com e sem o uso desse passo de compilação, bem como o percentual de melhoria resultante.

Ao contrário do esperado inicialmente, a melhoria obtida com o uso do *loop unrolling* é muito pequena, apesar de haver uma consistência entre as três estratégias de DSE. Analisando os resultados obtidos, foi possível observar que o *loop unrolling* foi bastante utilizado nas sequências evoluídas para as funções de referência (em 10 de 11 funções, exceto a *fdct*). Por outro lado, nas funções alvo, esse passo foi pouco empregado (ape-

Tabela 6.14: Média geométrica dos *speedups* obtidos pelas estratégias de DSE em função da utilização do *loop unrolling*.

	DSE-Limp	DSE-CI	DSE-AG
Sem <i>loop unrolling</i>	1,33	1,42	1,43
Com <i>loop unrolling</i>	1,36	1,45	1,47
Melhoria obtida	3%	2%	3%

nas em 5 de 46 funções). Acreditamos que isso ocorre devido a uma particularidade das funções presentes em cada conjunto, uma vez que o AG realiza o mesmo procedimento na busca das sequências, sem fazer distinção entre os dois conjuntos. Por isso, considerando a atual divisão das funções nos conjuntos referência/alvo, o uso do *loop unrolling* pouco acrescenta na melhoria do *speedup* das funções alvo, exceto para essas 5 funções (*autcor—*, *fdct—*, *goraud—*, *img_bounda* e *img_corr_g*).

A Figura 6.18 apresenta um gráfico com os melhores *speedups* alcançados individualmente para cada função alvo a partir dessas estratégias de DSE usando o *loop unrolling* no espaço de otimizações. Para facilitar a comparação, o gráfico também apresenta a média geométrica desses *speedups*.

Analisando o desempenho obtidos pelas estratégias de DSE usando o *loop unrolling*, podemos verificar um comportamento muito similar ao apresentado quando o passo de compilação não foi empregado. Em outras palavras, o DSE que adota apenas o algoritmo de limpeza apresenta um desempenho um pouco abaixo das demais estratégias, enquanto que o DSE-CI alcançou um desempenho muito próximo ao obtido pelo DSE-AG a partir do conjunto completo. Entretanto, em algumas funções, tais como: *goraud—* e *img_bounda*, houve um aumento significativo entre o melhor *speedup* atingido pelas abordagens baseadas no agrupamento e o DSE-AG considerando o espaço completo. Essa variação de desempenho foi causada pela aplicação repetida do *loop unrolling* na sequência gerada pelo algoritmo genético, enquanto que nas sequências obtidas nas demais abordagens, o passo foi utilizado uma única vez. Como já descrito anteriormente, os métodos de busca empregados nessas estratégias têm dificuldade em gerar sequências de otimização com repetição de passos de compilação.

Ao compararmos os desempenhos obtidos por uma mesma estratégia de DSE com e sem o *loop unrolling*, observamos que a presença do passo alterou algumas sequências do conjunto de referência. No DSE-Limp, tais mudanças acarretaram um desempenho fraco para as funções *dsp_mat_tr* (1,06) e *dsp_blk_mo* (1,00, ou seja, sem *speedup*), enquanto que no DSE-CI provocou uma redução de desempenho somente na *img_dilate* (de 1,38 para 1,35). Em contrapartida, houve uma melhoria no desempenho de 06 funções no DSE-CI e 16 funções na exploração através do DSE-Limp.

Figura 6.18: *Speedups* obtidos para cada função alvo através das estratégias de DSE não dominadas e usando o passo de compilação *loop unrolling*.

Diante dos resultados observados, consideramos que nossa abordagem tem potencial para lidar tanto com a ordenação dos passos de otimização, quanto com o problema da parametrização. Entretanto, para uma exploração mais adequada desse espaço de soluções mais complexo, se torna necessária a adoção de métodos de busca capazes de explorar sequências com repetições de passos, bem como considerar características dinâmicas do código que permita um agrupamento mais eficaz em relação ao desdobramento de laços.

6.4.8 Avaliação de novos *benchmarks*

Além das funções retiradas dos repositórios de *benchmark* da Texas Instruments (DSP e processamento de imagem), utilizamos nossa abordagem para explorar sequências de otimização para 4 funções adaptadas do repositório MiBench (Guthaus et al, 2001) (*quick sort*, *Dijkstra*, *blowfish* - codificador e decodificador); uma função *hot-spot* de uma aplicação industrial (filtro *Subband* - parte de um codificador de áudio MPEG), que foi concedido no contexto do projeto REFLECT (Cardoso et al, 2013); e 4 aplicações adaptadas do repositório UTDSP (Lee, 2002) (*compress*, *histogram*, *edge detection* e *lpc*). Os experimentos com as funções do MiBench e o filtro *Subband* foram realizados a fim de avaliar a efetividade da nossa abordagem em otimizar códigos de outros domínios de computação, ou seja, verificar o nível de generalização do conjunto de referência usado. A adoção de aplicativos do UTDSP visa avaliar o comportamento da nossa abordagem quando aplicado a códigos mais complexos.

A Figura 6.19 destaca os melhores *speedups* alcançados com as estratégias de DSE baseadas no algoritmo genético (DSE-AG), algoritmo de limpeza (DSE-Limp) e algoritmo de limpeza seguida de inserção (DSE-CI). A última coluna no gráfico da Figura 6.19 mostra as respectivas médias geométricas.

Figura 6.19: *Speedups* alcançados para cada novo *benchmark* usando as estratégias de DSE não-dominadas.

Como pode ser observado, as estratégias de DSE propiciaram uma melhoria no desempenho das funções/aplicações otimizadas. A única exceção foi a aplicação *lpc*, cujo

desempenho não foi melhorado por nenhuma das estratégias de exploração utilizadas. Acreditamos que isto ocorreu uma vez que suas duas principais funções (*lpc_analysis* e *sift*) foram compiladas separadamente, cada qual usando sua sequência de otimização específica, e os códigos otimizados foram combinados para a simulação. Apesar dessas explorações individuais alcançarem um pequeno *speedup* em relação ao código original (3% e 1%, respectivamente), esse ganho é perdido após a ligação (linkedição) dos códigos para a geração do código de máquina. No geral, o DSE-AG usando o conjunto completo alcançou os melhores resultados para todos os casos, enquanto que o algoritmo de limpeza obteve o pior desempenho dentre as estratégias testadas. O DSE-CI atingiu *speedups* muito próximos aos obtidos pelo algoritmo genético. A média geométrica dos *speedups* alcançados pelo DSE-AG completo, DSE-CI e DSE-Limp foram de $1.43\times$, $1.41\times$ e $1.28\times$, respectivamente. Esse comportamento é compatível com aquele observado durante a exploração das funções avaliadas inicialmente (*benchmarks* da TI).

Dado que os novos *benchmarks* são mais complexos que as funções utilizadas inicialmente, o tempo necessário para a sua exploração é bem maior que o exigido nos experimentos prévios. Entretanto, compatível com o que foi observado nas análises prévias, o DSE-Limp é a estratégia mais rápida e o DSE-AG com o espaço completo foi o mais lento. O tempo médio de execução alcançado pelo DSE-AG completo, DSE-CI e DSE-Limp foi de 27.778, 1.357 e 62 segundos, respectivamente. Portanto, nós concluímos que o panorama geral foi mantido mesmo usando aplicações mais complexas ou funções uade outros domínios: (i) DSE-AG atinge os melhores *speedups*, mas o DSE-CI retoma quase o mesmo desempenho com um processamento muito mais rápido; (ii) DSE-Limp retorna os piores *speedups*, mas ele é extremamente rápido.

6.4.9 Estendendo a análise comparativa

A adoção de diferentes ambientes de compilação e arquiteturas alvo torna difícil comparar os resultados entre trabalho correlatos. Para contornar esse problema, realizamos experimentos com um ambiente de compilação baseado na infraestrutura do compilador LLVM (versão 3.5) (Lattner e Adve, 2004), visando executar o código compilado no processador *softcore* LEON3 (Aeroflex, 2005). Tanto a infraestrutura do compilador LLVM, quanto o processador LEON3 são projetos de código aberto e amplamente utilizados em suas respectivas áreas. Isso permite a reprodução do ambiente de compilação, bem como possibilita a comparação com trabalhos que adotam o mesmo ambiente. Além disso, os *speedups* alcançados por nossas estratégias de DSE podem ser comparados com os resultados obtidos pelas estratégias de compilação -O (níveis de otimização) do próprio compilador. Neste trabalho, comparamos o desempenho da nossa abordagem com o nível de otimização -O2 e com o melhor *speedup* obtido pela exploração exaustiva das 10 melhores sequências de otimização apresentadas por Purini e Jain (2013), que é uma

abordagem baseada em aprendizado de máquina e considerada um estado-da-arte para o problema.

A Figura 6.20a apresenta as melhorias de desempenho obtidas através das estratégias DSE-Limp e DSE-CI em relação ao -O2 do LLVM. Comparando os *speedups* obtidos pelo DSE-CI e pelo -O2, observamos que 74% de todos os 54 códigos compilados alcançaram melhores desempenhos usando o DSE-CI, o -O2 foi superior em 17% deles; e os demais (9%) apresentaram os mesmos resultados para ambas as abordagens. A média geométrica dos *speedups* em relação ao -O2 alcançados pelo DSE-CI para os 54 *benchmarks* é de $1.07\times$. Se observamos apenas os 10, 20 e 30 melhores resultados, a média geométrica dos *speedups* obtidos pelo DSE-CI (linhas pretas) é de $1.49\times$, $1.32\times$ e $1.24\times$, respectivamente. Considerando os *speedups* obtidos através do DSE-Limp e do -O2, 54% dos códigos alcançaram melhores *speedups* usando o DSE-Limp, 37% usando o -O2, e 9% retornaram os mesmos resultados com as duas abordagens. A média geométrica dos *speedups* alcançados pelo DSE-Limp em relação ao -O2 é de 0,94. Quando consideramos apenas os 10, 20 e 30 melhores resultados, a média geométrica dos *speedups* obtidos pelo DSE-Limp (linhas cinza) é de $1.28\times$, $1.18\times$ e $1.13\times$, respectivamente. Portanto, podemos concluir que, no geral, o DSE-CI obteve melhores resultados do que o -O2 do compilador LLVM, corroborando o bom desempenho encontrado anteriormente com o compilador ReflectC. Entretanto, o DSE-Limp apresentou um desempenho pior, e seus resultados são semelhantes aos encontrados com a opção -O2. Acreditamos que a deterioração de desempenho do ReflectC para o LLVM pode ser justificada pelo fato do compilador LLVM empregar muito mais passos de otimização (124 contra 49) e alguns deles, como o desdobramento de laços (*loop unrolling*), possuem uma forte dependência semântica. Em contrapartida, o processo de agrupamento usado em nossa abordagem não considera informações semânticas, sendo totalmente guiado pelas estruturas sintáticas dos códigos.

A Figura 6.20b mostra o desempenho alcançado pelo uso das estratégias DSE-Limp e DSE-CI em relação à busca exaustiva sobre as 10 melhores sequências apresentadas por Purini e Jain (2013). Considerando o desempenho obtido com o uso do DSE-CI e o melhor *speedup* alcançado a partir das sequências de Purini (PS), 91% dos códigos compilados alcançaram melhores *speedups* com o DSE-CI, as PS superaram o DSE-CI em 6% dos códigos, e os outros 3% retornaram o mesmo *speedup* com ambas abordagens. A média geométrica dos *speedups* alcançados com o DSE-CI em relação às PS é de $1.24\times$. Se observarmos somente os 10, 20 e 30 melhores resultados, a média geométrica dos *speedups* alcançadas com o DSE-CI é de $2.11\times$, $1.63\times$ e $1.44\times$. Comparando o desempenho do DSE-Limp e das PS, a nossa abordagem foi superior para 69% dos códigos compilados, as PS obtiveram melhores resultados em 30% deles, e as abordagens foram similares em apenas 1%. A média geométrica dos *speedups* em relação às PS para todos os 54 códigos é de $1.09\times$. Considerando somente os 10, 20 e 30 melhores *speedups*, a média geomé-

trica obtida com o DSE-Limp é de $1,83\times$, $1,45\times$ e $1,31\times$. No geral, os experimentos com o compilador LLVM demonstraram que o uso da estratégia DSE-CI alcançou resultados muito melhores que as 10 melhores seqüências descritas por Purini e Jain (2013): mais de 90% dos códigos compilados foram melhorados em relação aos melhores *speedups* obtidos através das PS. A estratégia DSE-Limp também apresentou um bom desempenho e seus resultados superaram as PS em quase 70% dos códigos avaliados.

Figura 6.20: *Speedups* alcançados no ambiente de compilação do LLVM usando as estratégias DSE-Limp e DSE-CI.

Capítulo 7

Conclusões

A seleção da sequência de otimizações aplicadas pelo compilador pode ter um impacto significativo no desempenho do código compilado e é dependente da aplicação e da plataforma a ser utilizada. Portanto, a adoção de um esquema de exploração do espaço de projeto (DSE) rápido e eficiente para auxiliar os desenvolvedores de sistemas embarcados é de suma importância. Esta tese apresenta uma abordagem de DSE que adota um método de seleção baseado em agrupamento para reduzir o número de otimizações usadas durante a exploração de sequências de otimização. Nós analisamos diferentes alternativas de algoritmo de busca no DSE de modo a empregar nossa abordagem baseada em agrupamento. Experimentos mostraram que o uso de nossa abordagem resulta em uma significativa redução do espaço de busca, tornando possível a utilização de algoritmos de DSE mais simples. Por exemplo, abordagens baseadas no algoritmo de limpeza resultam em um bom compromisso entre o nível de otimização atingido pelas funções (redução da latência) e o tempo de execução do DSE. Particularmente, o *speedup* atingido são comparáveis com aqueles alcançados pelas melhores sequências de otimização (próximas do ótimo) obtidas na exploração com algoritmo genético (AG) sobre o conjunto completo de passos de otimização. Quando comparado com os resultados alcançados pela estratégia de DSE baseada em AG e visando o processador Xilinx MicroBlaze, nossa abordagem baseada no algoritmo mais simples e rápido (algoritmo de limpeza) permitiu uma significativa redução no tempo médio de exploração do DSE (em torno de 290 vezes), ao mesmo tempo que resultou em uma melhoria de desempenho apenas 10% menor que aquela obtida pelo AG, garantindo um *speedup* médio de 33% sobre o código sem otimização (*baseline*) para as 46 funções dos *benchmarks* da Texas Instruments, usados em nossos experimentos. O algoritmo de limpeza seguida de inserção (DSE-CI) resultou em um *speedup* médio similar ao encontrado pela abordagem baseada em AG (42% e 43%, respectivamente), com um tempo de execução do DSE 20 vezes menor (280 segundos comparado aos 92 minutos of AG).

Experimentos adicionais realizados com o compilador LLVM e o processador LEON3 demonstraram a importância da nossa abordagem mesmo quando usada em um ambiente

de compilação diferente daquele no qual foi originalmente desenvolvida. Quando comparado com os resultados obtidos com o nível de otimização -O2 do LLVM, nossa estratégia DSE-CI alcançou um desempenho médio superior, enquanto que o algoritmo de limpeza apresentou um desempenho similar. Especificamente, a nossa abordagem atingiu médias geométricas de *speedup* de $1,49\times$, $1,32\times$, e $1,24\times$, considerando, respectivamente, as 10, 20 e 30 melhores funções alvo. Considerando todos os 54 *benchmarks* avaliados, nossa abordagem propiciou atingiu uma melhoria de desempenho mais modesta (7%) em relação ao -O2.

Análises complementares também foram realizadas com o objetivo de avaliar se os componentes e as configurações empregados no método de seleção são adequados ou se eles poderiam ser melhor ajustados ao nosso propósito. Alguns aspectos do processo de seleção foram investigados mais detalhadamente durante os experimentos de modo a verificar sua influência na identificação de similaridades e dissimilaridades entre as funções manipuladas. Nesse processo, as seguintes questões foram investigadas: (i) A representação do código através do DNA é uma boa forma de entrada para o método de seleção baseado em agrupamento? (ii) As regras de transformação empregadas na codificação do DNA são adequadas para a identificação das similaridades entre as funções? (iii) O código DNA apenas dos laços (*loops*) é suficiente para representar os elementos realmente relevantes dos códigos fonte das funções? (iv) A métrica NCD e a biblioteca de compressão *bzlib* são a melhor combinação a ser usada para quantificar as similaridades/dissimilaridades entre as funções?

No geral, concluímos que as decisões e escolhas empregadas na implementação do método de seleção em nosso ambiente de DSE foram adequadas, sendo a especificação adotada uma das melhores possíveis, dentre as opções avaliadas. Algumas observações mais específicas foram concluídas em relação às questões acima: (i) O DSE a partir do DNA apresenta um desempenho similar ao DSE usando o *Assembly*. Entretanto, o tamanho do código DNA é bem menor que seu equivalente em *Assembly* (cerca de $15\times$), bem como sua geração é independente do compilador. O código fonte se mostrou inadequado para nossa abordagem, uma vez que não conseguiu agrupar funções similares. Portanto, consideramos que a representação em DNA é a mais apropriada para a nossa abordagem. (ii) As regras de transformação atualmente utilizadas na codificação do DNA são mais estáveis entre os conjuntos de regras avaliados, retornando os melhores *speedups* ou valores próximos deles. O comportamento apresentado pelo conjunto de regras adotado indica sua capacidade em representar, de forma satisfatória, as características dos códigos fonte de funções dos domínios de processamento digital de sinais, vídeo e imagens, especialmente na compilação de *software* visando a execução em processadores embarcados. Entretanto, acreditamos que várias das regras de DNA avaliadas nesta tese são importantes quando a otimização focar na síntese de aceleradores em *hardware*. (iii) As diferenças de desempenho resultante da delimitação do escopo de codificação do DNA foi muito sutil,

ou seja, a aplicação das regras de transformação somente sobre as estruturas dos *loops* ou em todo o código da função resultou em um comportamento similar, com uma pequena variação na média geométrica dos *speedups* alcançados. Entretanto, mesmo usando diferentes conjuntos de regras de transformação, a aplicação das regras apenas nos laços foi consistentemente melhor do que sobre todo o código. Acreditamos que isso ocorre porque, para os domínios de aplicação investigados, o processamento principal se encontra dentro dos laços. Portanto, considerando o conjunto de funções aqui investigado, a estratégia de codificar apenas as estruturas de *loops* parece ser a mais adequada. (iv) A Distância de Compressão Normalizada (NCD) alcançou os melhores *speedups* médios para as estratégias de DSE entre as métricas de distância avaliadas. Entretanto, outras distâncias baseadas na existência de símbolos e *substrings* também resultaram em um bom desempenho geral. Por outro lado, métricas baseadas em edição mostraram alguma dificuldade para identificar bons agrupamentos, embora tenham conseguido atingir valores próximos aos obtidos pelo DSE-AG completo para algumas funções individuais. Além disso, observamos que as bibliotecas de compressão avaliadas no cálculo da NCD não afetam de forma significativa o desempenho da nossa abordagem exploração.

Durante os experimentos, foi observada a importância da representatividade do conjunto de referência no desempenho da nossa abordagem de DSE baseada em agrupamento. Neste sentido, uma investigação foi conduzida a fim de definir um procedimento sistemático para a seleção de um conjunto de referência representativo a partir de um grupo de funções disponíveis. Na melhor estratégia de seleção, um agrupamento inicial é realizado, no qual todas as funções são agrupadas em k grupos, sendo k o tamanho especificado para o conjunto de referência (neste caso, usamos $k = 11$ para manter a relação com o conjunto originalmente adotado). Posteriormente, uma função de cada grupo é selecionada de modo a garantir a melhor cobertura dos passos de otimização disponibilizados pelo compilador. Esse procedimento de seleção resultou nos maiores desempenhos do DSE quando comparado com as demais estratégias investigadas, considerando as médias (aritmética e geométrica), a mediana, o pior e o melhor caso, bem como apresentou uma pequena variação entre os valores de máximo e mínimo dos *speedups* médios obtidos pelos conjuntos avaliados nessa estratégia. A partir do melhor conjunto de referência gerado por essa estratégia, o DSE-Limp atingiu um *speedup* médio de 1,40, o qual ficou bem próximo daqueles obtidos pelo DSE-CI (1,43) e pelo DSE-AG usando o conjunto completo de passos de otimização (1,44). Em relação ao desempenho alcançado pela mesma estratégia de DSE usando o conjunto de referência original, houve uma melhoria no *speedup* de 7%. Além disso, o desempenho do pior conjunto gerado por essa abordagem é superior àquele obtido com conjunto de referência original (1,33), indicando a capacidade da estratégia em definir conjuntos de referência adequados aos propósitos de nossa abordagem.

7.1 Contribuições

As principais contribuições de nossa pesquisa são:

1. Desenvolvimento de uma nova abordagem de exploração de sequências de otimização, com as seguintes características:
 - Exploração baseada em métodos simples de busca aplicados sobre um conjunto reduzido de otimizações, os quais não seriam eficientes se aplicados sobre o espaço completo. A propriedade determinística dos métodos simples e híbridos baseados nos algoritmos de limpeza e inserção garante que sempre a mesma sequência será gerada a partir de uma mesma entrada. Através de uma análise multi-objetivos da eficácia das estratégias, verificamos que até o método mais simples, baseado no algoritmo de limpeza, faz parte da fronteira de não-dominância, por retornar um *speedup* razoável a um baixo custo de tempo. Por outro lado, a estratégia híbrida que utiliza limpeza seguida de inserção também faz parte dessa fronteira por ser o método mais equilibrado, retornando um excelente *speedup* (próximo da melhor referência) a um custo de tempo razoável.
 - Uso de funções previamente otimizadas como um histórico para a determinação de um espaço de busca reduzido e promissor. Essa capacidade da nossa abordagem propicia a utilização de experiências de projetos anteriores de forma implícita, ou seja, "escondidas" dentro do histórico. Esse conjunto de referência que representa o histórico pode ser aumentado incrementalmente com novas funções a fim de melhorar a representatividade do conjunto e, consequentemente, dos agrupamentos gerados. Uma das principais vantagens dessa abordagem é ser não supervisionada, ou seja, não demanda qualquer etapa de processamento *off-line* para o treinamento ou modelagem. Sua aprendizagem é baseada unicamente nos exemplos fornecidos pelo conjunto de referência (conhecimento dinâmico).
 - Seleção de passos de otimização sem a necessidade de qualquer conhecimento prévio sobre os passos, sua ordem, ou a etapa de sua aplicação pelo compilador. Isso é particularmente útil para os compiladores modernos, os quais são constantemente atualizados, com a inclusão de novos passos de compilação, dificultando o aprimoramento das regras aplicadas na otimização do código.
 - Obtenção de estratégias de DSE para a exploração rápida de sequências de otimização que resultem em um bom desempenho para uma função específica.
 - Independência da linguagem e da plataforma utilizadas, possibilitando sua fácil adaptação a outras arquiteturas e/ou projetos. As especificidades de cada

linguagem ou plataforma são capturadas informalmente através das funções existentes no conjunto de referência.

2. Contribuições computacionais em como extrair informações relevantes de resultados previamente explorados para um conjunto de funções de referência a fim de realizar a exploração de novas funções alvo. Inicialmente, investigamos o uso de diferentes representações de entrada e concluímos que o DNA é a melhor opção para nossa abordagem. Em seguida, avaliamos o efeito da adoção de diferentes regras de transformação para a codificação do DNA, bem como do escopo de aplicação dessas regras (todo programa ou somente nas estruturas de *loops*). A partir dos resultados, concluímos que as regras atuais aplicadas sobre as estruturas dos *loops* são suficientes para a identificação de sequências de otimização de componentes de *software*. Entretanto, acreditamos que em compilações direcionadas a aceleradores de *hardware*, algumas das regras avaliadas, tais como, representação de tipos de dados e operações aritméticas, são essenciais para uma otimização adequada. Experimentos em *hardware* estão fora do escopo desse trabalho, ficando essa análise como sugestão para trabalhos futuros. Uma vez definida a fonte de entrada do método de seleção, foi realizada a análise dos elementos envolvidos no cálculo das matrizes de dissimilaridades. Nesse processo, diferentes métricas de distância foram avaliadas a fim de se determinar aquelas que melhor identificam as dissimilaridades entre os códigos das funções. Foram avaliadas a distância NCD, várias métricas de distância entre *strings* encontradas em um pacote da linguagem R, e duas novas implementadas neste trabalho (DCI e DDN), que não consideram a ordem dos símbolos no DNA. Os resultados indicaram que a NCD é uma métrica de distância adequada à nossa abordagem, capaz de alcançar os melhores valores de *speedup* médio. Métricas baseadas em granularidade são mais adequadas ao nosso método do que aquelas baseadas em edição. Além disso, verificamos que a disposição dos símbolos no DNA não influencia significativamente nos agrupamentos, portanto, algumas métricas de granularidade avaliadas conseguiram um desempenho similar ao da NCD. A NCD e a DDN são as que melhor atendem às necessidades do nosso método, dado que resultaram nos melhores *speedups* médios. Entretanto, acreditamos que a combinação de um grupo de métricas de distância possa ser promissor. Diferentes bibliotecas de compressão também foram testadas para avaliar como elas impactavam no cálculo da matriz de distâncias através da NCD. Entretanto, com exceção da PPMDX, todas as demais resultaram em valores muito similares, sendo possível adotá-las em nossa abordagem sem prejuízo à precisão dos resultados.
3. Definição de critérios para a seleção do conjunto de referência. Experimentos mostraram a importância da utilização de bons históricos como conjunto de referência, constatando que sua representatividade pode beneficiar significativamente a explo-

ração das sequências de otimização que resultem no melhor desempenho da função alvo. A partir desse entendimento, foram concebidos e avaliados alguns critérios para identificar as funções relevantes para o agrupamento, com o objetivo de definir um método de seleção do conjunto de referência. A partir da formalização sistemática desse processo de seleção das funções de referência, torna-se possível a criação de diferentes conjuntos de acordo com o domínio da aplicação a ser explorado e a plataforma almejada, bem como desenvolver uma metodologia para o crescimento incremental desse conjunto, de modo a mantê-lo o mais representativo possível.

4. Proposição de um novo paradigma baseado em agrupamento para diferentes etapas de otimização no projeto de sistemas embarcados. Apesar do escopo deste trabalho ser relacionado à otimização de funções para microprocessadores embarcados, acreditamos que nossa abordagem de exploração baseada em agrupamento possui um grande potencial de utilização em outras etapas do projeto de *software* e *hardware* para sistemas embarcados. Por exemplo, uma abordagem similar poderia ser conduzida na etapa de otimização de aceleradores em *hardware* ou em projetos *multicore*.

7.2 Artigos Originados da Tese

Partes dessa pesquisa geraram a publicação de 2 artigos (Martins et al, 2014b) (Martins et al, 2014a) em eventos científicos internacionais.

Em Martins et al (2014b), exploramos a seleção prévia dos passos de otimização para reduzir o espaço da busca através de nossa abordagem baseada em agrupamento e apresentamos novas estratégias de DSE baseadas em algoritmos de busca simples capazes de atingir resultados de *speedup* similares aos obtidos pela exploração de todo o espaço de otimizações através de Algoritmos Genéticos com muito menos tempo de exploração.

Em Martins et al (2014a), focamos na nossa abordagem de seleção de otimizações baseada em agrupamento, suas propriedades e detalhes, e avaliamos se as melhorias alcançadas na exploração das sequências eram decorrentes do agrupamento, da redução no espaço de busca ou de ambos.

Além dessas publicações, também foi submetido, aprovado e encontra-se em fase de publicação, um artigo para o periódico internacional *ACM Transactions on Architecture and Code Optimization (TACO)* (Martins et al, 2015).

Em Martins et al (2015), descrevemos os efeitos da configuração adotada para o método de seleção na qualidade dos resultados alcançados pelas nossas estratégias de DSE, bem como investigamos o impacto da adoção de diferentes critérios de seleção das funções para o conjunto de referência.

7.3 Trabalhos Futuros

Como pode ser observado, a representação de DNA adotada nesta tese é totalmente baseada em regras de transformações orientadas à sintaxe. A ideia é verificar se esta abordagem simples é suficiente para obtermos agrupamentos eficientes. Entretanto, ainda é necessário investigar o impacto da adoção de outros tipos de regras de transformação, como aquelas semanticamente orientadas ou que incluem informação de dependência de dados. Essas últimas devem influenciar significativamente na escolha das otimizações baseadas em paralelismo, como o *loop unrolling*.

Os resultados obtidos a partir do uso do *loop unrolling* não foram conclusivos, dado que poucas funções alvo utilizaram esse passo de otimização, enquanto praticamente todas as funções de referência adotaram pelo menos um desdobramento. Por tanto, é premente a repetição desses experimentos considerando uma nova distribuição das funções entre os conjuntos de referência e alvo, garantindo uma melhor representatividade dos dois cenários. Entretanto, esses resultados preliminares foram suficientes para identificar a necessidade de adaptação dos métodos determinísticos de busca baseados em limpeza e inserção (híbridos e simples) atuais, ou a investigação de novos métodos capazes de gerar sequências de otimização com repetição de passos de compilação ao mesmo tempo que explora diferentes configurações de *loop unrolling*, sem onerar o custo da exploração.

O estudo de diferentes métricas para calcular a distância entre os DNAs mostrou que, em termos individuais, a métrica NCD atende adequadamente às necessidades do nosso método de seleção baseado em agrupamento, uma vez que nenhuma outra métrica conseguiu melhorar o desempenho do DSE em termos de *speedup* médio. Entretanto, através da análise dos resultados individuais de cada função alvo, observamos a ocorrência de oscilações de desempenho, mesmo para as distâncias que apresentam o pior desempenho global. Assim, acreditamos que mais de uma métrica possa ser utilizada, a fim de criar uma floresta de árvores filogenéticas, as quais poderiam ser utilizadas em uma espécie de comitê para definir o agrupamento por consenso ou maioria. Essa ideia é similar ao conceito de *ensemble*, que visa melhorar a acurácia das classificações ou regressões através da combinação dos resultados de um grupo de modelos mais fracos (Faceli et al, 2011). Essa é uma abordagem interessante, que deve ser investigada a fim de determinar se o custo computacional decorrente do processo de votação será compensado pelo ganho na precisão da exploração.

Neste trabalho avaliamos diferentes critérios de seleção que possibilitem a escolha sistemática do conjunto de referência. Essa investigação foi limitada a conjuntos de mesmo tamanho, o que permitiu simplificar o espaço de busca e facilitou a comparação do desempenho dos novos conjuntos em relação àquele originalmente adotado. Entretanto, uma exploração mais ampla é essencial para um melhor entendimento do comportamento desse conjunto, de modo que possa ser avaliada a sua representatividade em função da

quantidade de funções utilizadas. Além disso, novos estudos devem ser realizados com o objetivo de definir bons critérios para a inclusão incremental de novas funções durante o ciclo de vida do método. Essas inclusões devem ser eficazes, de modo a melhorar o desempenho da exploração sem que o conjunto se torne tendencioso. Um extensão natural deste trabalho é o desenvolvimento de uma ferramenta a ser incorporada ao ambiente de compilação que seja capaz de selecionar, de forma automática, funções representativas para formar um novo conjunto de referência, de acordo com a métrica de avaliação escolhida, bem como identificar novas funções que aumentariam a cobertura/abrangência de um conjunto de referência já existente.

Outro ponto a ser explorado é o uso de técnicas de aprendizagem de máquina, tais como redes bayesianas, para extrair informações relevantes sobre o posicionamento dos passos nas sequências de otimização das funções de referência. As informações sobre a distribuição dos passos de otimização podem então ser utilizadas para guiar os algoritmos de busca de modo mais eficiente e efetivo.

Uma extensão natural desta pesquisa é aplicar a nossa abordagem na exploração de sequências de otimizações no contexto de geração de *hardware* para os FPGAs e/ou sistemas *multicore*, visando ampliar as possibilidades de exploração através de técnicas de paralelismo como *loop pipelining*, bem como de parâmetros arquiteturais, por exemplo, a configuração do tamanho da cache.

Por fim, o aumento na densidade dos *chips* e na frequência de processamento faz com que a potência dissipada pelos circuitos digitais aumentasse exponencialmente (Ma, 2007). Assim, o consumo de energia vem se tornando um dos principais requisitos para sistemas embarcados de alto desempenho, sendo utilizado como critério de otimização no projeto dos mesmos. A redução no consumo de energia é especialmente requerida em aplicações móveis, uma vez que a tecnologia das baterias tem evoluído mais lentamente e o custo da energia é alto (Marwedel, 2010). Minimizar o consumo de energia também reduz a necessidade de resfriamento e, conseqüentemente, melhora a confiabilidade do sistema, dado que o tempo de vida dos circuitos eletrônicos diminui com as altas temperaturas. Nesse sentido, é fundamental conhecer o consumo da arquitetura proposta durante sua operação. Portanto, é fundamental guiar a busca por uma arquitetura balanceada entre desempenho e consumo de energia. Para isso, é necessário o desenvolvimento de estimadores rápidos e suficientemente precisos a partir de uma representação de alto nível do sistema. O principal objetivo desses estimadores é possibilitar a avaliação do impacto das escolhas de arquitetura e compilação nos estágios iniciais do desenvolvimento de uma solução, possibilitando uma maior cobertura do espaço de projeto. Assim, como trabalho futuro pretendemos desenvolver estimadores baseados no consumo das instruções do processador (Al-Khatib e Abdi, 2015), a fim de se obter estimações rápidas e com um grau de acurácia que permita diferenciar adequadamente a eficiência de cada solução candidata.

Referências

ACE Cosy compiler development system. 2014.

Disponível em: <http://www.ace.nl/compiler/cosy.html>

AEROFLEX Leon3 processor. 2005.

Disponível em: <http://www.gaisler.com/index.php/products/processors/leon3>

AGAKOV, F.; BONILLA, E.; CAVAZOS, J.; FRANKE, B.; FURSIN, G.; O'BOYLE, M. F. P.; THOMSON, J.; TOUSSAINT, M.; WILLIAMS, C. K. I. Using machine learning to focus iterative optimization. In: *Proceedings of the Int. Symp. on Code Generation and Optimization (CGO'06)*, 2006, pp. 295–305.

AL-KHATIB, Z.; ABDI, S. Operand-value-based modeling of dynamic energy consumption of soft processors in fpga. In: SANO, K.; SOUDRIS, D.; HÜBNER, M.; DINIZ, P. C., eds. *Applied Reconfigurable Computing*, vol. 9040 of *LNCS*, Springer International Publishing, pp. 65–76, 2015.

ALMAGOR, L.; COOPER, K. D.; GROSUL, A.; HARVEY, T. J.; REEVES, S. W.; SUBRAMANIAN, D.; TORCZON, L.; WATERMAN, T. Finding effective compilation sequences. In: *Proceedings of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, 2004, pp. 231–239.

ALT, M.; ASSMANN, U.; SOMEREN, H. Cosy compiler phase embedding with the cosy compiler model. In: *Proceedings of the 5th Int. Conf. on Compiler Construction (CC'94)*, London, UK, UK: Springer-Verlag, 1994, pp. 278–293.

ASCIA, G.; CATANIA, V.; NUOVO, A. G. D.; PALESI, M.; PATTI, D. Performance evaluation of efficient multi-objective evolutionary algorithms for design space exploration of embedded computer systems. *Appl. Soft Comput.*, vol. 11, pp. 382–398, 2011.

ASCIA, G.; CATANIA, V.; PALESI, M. A ga-based design space exploration framework for parameterized system-on-a-chip platforms. *IEEE Trans. on Evolutionary Computation*, vol. 8, no. 4, pp. 329–346, 2004.

- ASHOURI, A. H.; MARIANI, G.; PALERMO, G.; SILVANO, C. A bayesian network approach for compiler auto-tuning for embedded processors. In: *IEEE 12th Symp. on Embedded Systems for Real-time Multimedia (ESTIMedia 2014)*, 2014, pp. 90–97.
- AUSLANDER, M.; HOPKINS, M. An overview of the pl.8 compiler. In: *Proceedings of the ACM SIGPLAN Notices Symposium on Compiler Construction*, (appud in Kulkarni (2007)), 1982, pp. 22–31.
- BECKER, J.; HUEBNER, M.; ULLMANN, M. Power estimation and power measurement of xilinx virtex fpgas: Trade-offs and limitations. In: *16th Symposium on Integrated Circuits and Systems Design, SBCCI'03*, Washington, DC, USA: IEEE Computer Society, 2003, pp. 283–288 (*SBCCI'03*,).
- BEG, M. O. *Combinatorial problems in compiler optimization*. Phd thesis, University of Waterloo, Ontario, Canadá, 2013.
- BELTRAME, G.; FOSSATI, L.; SCIUTO, D. Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 1083–1095, 2010.
- BENITEZ, M. E.; DAVIDSON, J. W. A portable global optimizer and linker. In: *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, (appud in Kulkarni (2007)), 1988, pp. 329–338.
- BOUDEC, J. Y. L.; THIRAN, P. *Network calculus: A theory of deterministic queuing systems for the internet*. Springer-Verlag, 2001.
- BRUNI, D.; BOGLIOLO, A.; BENINI, L. Statistical design space exploration for application-specific unit synthesis. In: *38th annual Design Automation Conference, DAC'01*, New York, NY, USA: ACM, 2001, pp. 641–646 (*DAC'01*,).
- BUYUKKURT, B.; CORTES, J.; VILLARREAL, J.; NAJJAR, W. A. Impact of high-level transformations within the roccc framework. *ACM Trans. on Architecture Code Optimization*, vol. 7, no. 4, pp. 1–36, 2010.
- CARDOSO, J. M. P.; CARVALHO, T.; COUTINHO, J. G. F.; LUK, W.; NOBRE, R.; DINIZ, P.; PETROV, Z. M. Lara: an aspect-oriented programming language for embedded systems. In: *Proceedings of the 11th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'12)*, 2012, pp. 179–190.
- CARDOSO, J. M. P.; DINIZ, P.; WEINHARDT, M. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys*, vol. 42, no. 4, pp. 13:1–13:65, 2010.
- CARDOSO, J. M. P.; DINIZ, P. C. *Compilation techniques for reconfigurable architectures*. Springer, 2008.

- CARDOSO, J. M. P.; DINIZ, P. C.; COUTINHO, J. G. F.; PETROV, Z. M., eds. *Compilation and synthesis for embedded reconfigurable systems: An aspect-oriented approach*. Springer, 2013.
- CAVAZOS, J.; FURSIN, G.; AGAKOV, F.; BONILLA, E.; O'BOYLE, M. F. P.; TEMAM, O. Rapidly selecting good compiler optimizations using performance counters. In: *Proceedings of the Int. Symposium on Code Generation and Optimization (CGO'07)*, 2007, pp. 185–197.
- CAVAZOS, J.; O'BOYLE, M. F. P. Method-specific dynamic compilation using logistic regression. In: *Proceedings of the ACM Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, 2006, pp. 229–240.
- CHABBI, M. M.; MELLOR-CRUMMEY, J. M.; COOPER, K. D. Efficiently exploring compiler optimization sequences with pairwise pruning. In: *Proceedings of the 1st Int. Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*, 2011, pp. 34–45.
- CHEN, Y.; FANG, S.; HUANG, Y.; EECKHOUT, L.; FURSIN, G.; TEMAM, O.; WU, C. Deconstructing iterative optimization. *ACM Trans. on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, pp. 21:1–21:30, 2012.
- CILIBRASI, A. R.; VITANYI, A. P. Clustering by compression. In: *Proceedings of the IEEE Trans. Information Theory (AOSD'12)*, 2005, pp. 1523–1545.
- CILIBRASI, R.; CRUZ, A.; ROOIJ, S.; KEIJZER, M. Complearn toolkit. 2008.
Disponível em: <http://www.complearn.org/>
- COOPER, K. D.; GROSUL, A.; HARVEY, T. J.; REEVES, S.; SUBRAMANIAN, D.; TORCZON, L.; WATERMAN, T. Exploring the structure of the space of compilation sequences using randomized search algorithms. *Journal of Supercomputing*, vol. 36, no. 2, pp. 135–151, 2006.
- COOPER, K. D.; SCHIELKE, P. J.; SUBRAMANIAN, D. Optimizing for reduced code space using genetic algorithms. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, 1999, pp. 1–9.
- COOPER, K. D.; SUBRAMANIAN, D.; TORCZON, L. Adaptive optimizing compilers for the 21st century. *J. Supercomputing*, vol. 23, no. 1, pp. 7–22, 2002.
- DINIZ, P. C.; CARDOSO, J. M. P. Code transformations for embedded reconfigurable computing architectures. In: FERNANDES, J. M.; LÄMMEL, R.; VISSER, J.; SARAIVA, J., eds. *Generative and Transformational Techniques in Software Engineering III*, vol. 6491 of LNCS 6491, Springer Berlin Heidelberg, pp. 322–344, 2011.

- DRAGOMIR, O. S. *K-loops: Loop transformations for reconfigurable architectures*. Phd thesis, TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica, Delft, Netherlands, 2011.
- DUBACH, C. *Using machine-learning to efficiently explore the architecture/compiler co-design space*. Phd thesis, University of Edinburgh - School of Informatics, Edinburgh, 2009.
- EYERMAN, S.; EECKHOUT, L.; BOSSCHERE, K. Efficient design space exploration of high performance embedded out-of-order processors. In: *Conf. on Design, Automation and Test in Europe, DATE'06*, European Design and Automation Association, 2006, pp. 351–356 (*DATE'06*,).
- FACELI, K.; LORENA, A. C.; GAMA, J. M. P.; CARVALHO, A. C. P. L. F. *Inteligência artificial: uma abordagem de aprendizado de máquina*. 1a edn.. Rio de Janeiro: LTC, 2011.
- FELSENSTEIN, J. *Inferring phylogenies*. Sinauer Associates, Inc., 2003.
- FERRANDI, F.; LANZI, P. L.; LOIACONO, D.; PILATO, C.; SCIUTO, D. A multi-objective genetic algorithm for design space exploration in high-level synthesis. In: *Proc. of the IEEE Computer Society Annual Symposium on VLSI, ISVLSI'08*, IEEE Computer Society, 2008, pp. 417–422 (*ISVLSI'08*,).
- FLEMING, P. J.; WALLACE, J. J. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986.
- FURSIN, G.; KASHNIKOV, Y.; MEMON, A.; CHAMSKI, Z.; TEMAM, O.; NAMOLARU, M.; YOM-TOV, E.; MENDELSON, B.; ZAKS, A.; COURTOIS, E.; BODIN, F.; BARNARD, P.; ASHTON, E.; BONILLA, E.; THOMSON, J.; WILLIAMS, C.; O'BOYLE, M. Milepost gcc: machine learning enabled self-tuning compiler. *Int. Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.
- GAJSKI, D. D.; ABDI, S.; GERSTLAUER, A.; SCHIRNER, G. *Embedded system design: Modeling, synthesis and verification*. 1st edn.. Springer, 2009.
- GANSSLE, J. *The art of programming embedded systems*. Elsevier Science, 2012.
- GENSER, A.; BACHMANN, C.; HAID, J.; STEGER, C.; WEISS, R. An emulation-based real-time power profiling unit for embedded software. In: *the 9th international conference on Systems, architectures, modeling and simulation, SAMOS'09*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 67–73 (*SAMOS'09*,).

- GIVARGIS, T.; VAHID, F.; HENKEL, J. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. *IEEE Transactions on Very Large Scale Integration Systems*, vol. 10, no. 4, pp. 416–422, 2002.
- GOLDBERG, D. E. *Genetic algorithms in search, optimization and machine learning*. 1st ed. edn.. Addison-Wesley Longman, 1989.
- GOURAUD, H. Continuous shading of curved surfaces. *IEEE Trans. on Computers*, vol. 20, no. 6, pp. 623–629, 1971.
- GRIES, M. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131–183, 2004.
- GUTHAUS, M. R.; RINGENBERG, J. S.; ERNST, D.; AUSTIN, T. M.; MUDGE, T.; BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite. In: *Proceedings of the IEEE Int. Workshop of the Workload Characterization (WWC'01)*, 2001, pp. 3–14.
- HANEDA, M.; KNIJNENBURG, P. M. W.; WIJSHOFF, H. A. G. Optimizing general purpose compiler optimization. In: *Proceedings of the 2nd Conf. on Computing frontiers (CF'05)*, 2005, pp. 180–188.
- HOSTE, K.; EECKHOUT, L. Cole: compiler optimization level exploration. In: *Proceedings of the 6th Int. Symp. on Code Generation and Optimization (CGO'08)*, 2008, pp. 165–174.
- HUANG, Q.; LIAN, R.; CANIS, A.; CHOI, J.; XI, R.; BROWN, S.; ANDERSON, J. The effect of compiler optimizations on high-level synthesis for fpgas. In: *Proceedings of the IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM'13)*, 2013, pp. 89–96.
- ITO, K.; ZEUGMANN, T.; ZHU, Y. Algorithms and applications. chap. Clustering the Normalized Compression Distance for Influenza Virus Data, Springer-Verlag, pp. 130–146, 2010.
- JANTZ, M. R. *Understanding optimization phase interactions to reduce the phase order search space*. master thesis, University of Kansas, School of Electrical Engineering and Computer Science, Kansas, USA, 2010.
- JANTZ, M. R.; KULKARNI, P. A. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In: *Proceedings of the 2013 Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES'13)*, 2013, pp. 7:1–7:10.
- JOHNSON, N. E. *Code size optimization for embedded processors*. Phd thesis, University of Cambridge, Cambridge, UK, 2004.

- KISUKI, T.; KNIJNENBURG, P. M. W.; O'BOYLE, M. F. P.; BODIN, F.; WIJSHOFF, H. A. G. A feasibility study in iterative compilation. In: POLYCHRONOPOULOS, C.; FUKUDA, K.; TOMITA, S., eds. *High Performance Computing*, vol. 1615 of *LNCS*, Springer Berlin Heidelberg, pp. 121–132, 1999.
- KLENK, S.; THOM, D.; HEIDEMANN, G. The normalized compression distance as a distance measure in entity identification. In: *Proceedings of the 9th Industrial Conf. on Advances in Data Mining (ICDM'09)*, 2009, pp. 325–337.
- KULKARNI, P. A. *Fast and effective solutions to the phase ordering problem*. Phd thesis, Florida State University, Department of Computer Science, Florida, USA, 2007.
- KULKARNI, P. A.; JANTZ, M. R.; WHALLEY, D. B. Improving both the performance benefits and speed of optimization phase sequence searches. In: *Proceedings of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'10)*, 2010, pp. 95–104.
- KULKARNI, P. A.; WHALLEY, D. B.; TYSON, G. S. Evaluating heuristic optimization phase order search algorithms. In: *Proceedings of the IEEE Int. Symp. on Code Generation and Optimization (CGO'07)*, 2007, pp. 157–169.
- KULKARNI, P. A.; WHALLEY, D. B.; TYSON, G. S.; DAVIDSON, J. W. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. on Architecture and Code Optimization (TACO)*, vol. 6, no. 1, pp. 1:1–1:36, 2009.
- KULKARNI, S.; CAVAZOS, J. Mitigating the compiler optimization phase-ordering problem using machine learning. In: *Proceedings of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, 2012, pp. 147–162.
- LAHIRI, K.; RAGHUNATHAN, A.; DEY, S. Efficient exploration of the soc communication architecture design space. In: *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD'00)*, 2000, pp. 424–430.
- LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: *Proc. of the Int. Symp. on Code Generation and Optimization (CGO'04)*, 2004.
- LEE, C. G. Utdsp benchmark suite. 2002.
Disponível em: <http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.tar.gz>
- LI, M.; VITANYI, P. M. B. *An introduction to kolmogorov complexity and its applications*. 2nd ed. edn.. Springer-Verlag, 1997.

- LIMA, E. D.; S. XAVIER, T. C.; SILVA, A. F.; RUIZ, L. B. Compiling for performance and power efficiency. In: *Proceedings of the 23rd Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'13)*, 2013, pp. 142–149.
- MA, J. T. H. *Estimativa de consumo de energia em nível de instrução para processadores modelados em archc*. dissertação de mestrado, Universidade Estadual de Campinas - Instituto de Computação, Campinas, SP, 2007.
Disponível em: <http://www.bibliotecadigital.unicamp.br/document/?code=vtls000436129>
- MANCHESTER, T. U. Mitoobox for c and matlab. 2014.
Disponível em: <http://www.cs.man.ac.uk/~pococka4/MITToolbox.html>
- MARTINS, L. G. A.; NOBRE, R.; CARDOSO, J. M. P.; DELBEM, A. C. B.; MARQUES, E. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Trans. on Architecture and Code Optimization (TACO)*, (em publicação), 2015.
- MARTINS, L. G. A.; NOBRE, R.; DELBEM, A. C. B.; MARQUES, E.; CARDOSO, J. M. P. A clustering-based approach for exploring sequences of compiler optimizations. In: *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2014)*, 2014a, pp. 1–8.
- MARTINS, L. G. A.; NOBRE, R.; DELBEM, A. C. B.; MARQUES, E.; CARDOSO, J. M. P. Exploration of compiler optimization sequences using clustering-based selection. In: *Proceedings of the ACM SIGPLAN/SIGBED conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'14)*, 2014b, pp. 63–72.
- MARWEDEL, P. *Embedded system design: Embedded systems foundations of cyber-physical systems*. 2nd edn.. Springer, 2010.
- MEEUWS, R. J.; GALUZZI, C.; BERTELS, K. L. M. High level quantitative hardware prediction modeling using statistical methods. In: *Proceedings of the Int. Conf. on Embedded Computer Systems: Architectures, Models, and Simulations (SAMOS'10)*, 2011, pp. 140–149.
- MENOTTI, R. *Lalp: uma linguagem para exploração do paralelismo de loops em computação reconfigurável*. tese de doutorado, Universidade de São Paulo - Instituto de Ciências Matemáticas e de Computação, São Carlos, SP, 2010.
Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-17082010-151100/>
- MITCHELL, M. *An introduction to genetic algorithms*. Cambridge, MA, USA: MIT Press, 1996.

- MOHANTY, S.; PRASANNA, V. K.; NEEMA, S.; DAVIS, J. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 18–27, 2002.
- MUCHNICK, S. S. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- NEWMAN, M. *Networks: An introduction*. Oxford Univ. Press, 2010.
- PALERMO, G.; SILVANO, C.; ZACCARIA, V. Multi-objective design space exploration of embedded systems. *Embedded Computing*, pp. 305–316, 2005.
- PALERMO, G.; SILVANO, C.; ZACCARIA, V. Respir: a response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1816–1829, 2009.
- PAN, Z.; EIGENMANN, R. Peak: a fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. on Programming Languages and Systems*, vol. 30, no. 3, pp. 1–17, 2008.
- PATTERSON, D. A.; HENNESSY, J. L. *Arquitetura de computadores: uma abordagem quantitativa*. 3a edn.. CAMPUS, 2003.
- PISCITELLI, R.; PIMENTEL, A. D. Design space pruning through hybrid analysis in system-level design space exploration. In: *Design, Automation & Test in Europe Conference & Exhibition DATE, DATE2012*, 2012, pp. 781–786 (DATE2012,).
- PURINI, S.; JAIN, L. Finding good optimization sequences covering program space. *ACM Trans. on Architecture and Code Optimatizion*, vol. 9, no. 4, pp. 56:1–56:23, 2013.
- QUÉVA, M. *Phase-ordering in optimizing compilers*. master thesis, Technical University of Denmark,, Denmark, 2007.
Disponível em: <http://etd.dtu.dk/thesis/259692/imm5406.pdf>
- RICHTER, K.; ZIEGENBEIN, D.; JERSAK, M.; ERNST, R. Bottom-up performance analysis of hw/sw platforms. In: *Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems (DIPES'02)*, (appud in Gries (2004)), 2002, pp. 173–183.
- ROY, C. K.; CORDY, J. R.; KOSCHKE, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

- SANCHES, A. K.; CARDOSO, J. M. P. Pattern-mining over repositories of benchmarks to identify and suggest reconfigurable functional units: A preliminary step. In: *Proceedings of Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2008)*, 2008, pp. 227–230.
- SANCHES, A. K.; CARDOSO, J. M. P. On identifying patterns in code repositories to assist the generation of hardware templates. In: *Proceedings of the 20th Int. Conf. on Field Programmable Logic and Applications (FPL'10)*, 2010, pp. 267–270.
- SANCHES, A. K.; CARDOSO, J. M. P.; DELBEM, A. C. B. Identifying merge-beneficial software kernels for hardware implementation. In: *Proceedings of the Int. conf. on Reconfigurable Computing and FPGAs (ReConFig'11)*, 2011, pp. 74–79.
- SANCHEZ, R.; AMARAL, J.; SZAFRON, D.; PIRVU, M.; STOODLEY, M. Using machines to learn method-specific compilation strategies. In: *Proceedings of the Int. Symp. on Code Generation and Optimization (CGO'11)*, 2011, pp. 257–266.
- SHER, G.; MARTIN, K.; DECHEV, D. Preliminary results for neuroevolutionary optimization phase order generation for static compilation. In: *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems (ODES'14)*, 2014, pp. 33–40.
- SHURMAN, J.; MOEHRING, M. Phylip toolkit for inferring phylogenies. 1980.
Disponível em: <http://evolution.genetics.washington.edu/phylip.html>
- SIGDEL, K. *System-level design space exploration of reconfigurable architectures*. Phd thesis, Delft University of Technology, 2011.
Disponível em: <http://repository.tudelft.nl/assets/uuid:f36ef6c3-0742-4372-a2f1-7f34f686ded6/Main.pdf>
- SILVA, A. E. A. *Uma abordagem multi-objetivo e multimodal para reconstrução de Árvores filogenéticas*. tese de doutorado, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas, São Paulo, Brasil, 2007.
- SILVANO, C.; FORNACIARI, W.; PALERMO, G.; ZACCARIA, V.; CASTRO, F.; MARTINEZ, M.; BOCCHIO, S.; ZAFALON, R.; AVASARE, P.; VANMEERBEECK, G.; YKMAN-COUVREUR, C.; WOUTERS, M.; KAVKA, C.; ONESTI, L.; TURCO, A.; BONDI, U.; MARIANI, G.; POSADAS, H.; VILLAR, E.; WU, C.; FAN, D.; HAO, Z.; TANG, S. Multicube: Multi-objective design space exploration of multi-core architectures. In: *Annual Symposium on VLSI, ISVLSI 2010*, IEEE Computer Society, 2010, pp. 488–493 (*ISVLSI 2010*,).
- SIPSER, M. *Introdução à teoria da computação*. 2a ed. edn.. São Paulo: Thomson Learning, 2007.

- SRINIVAS, N.; DEB, K. Multiobjective optimization using nondominated sorting in genetic algorithms. *Journal Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, 1994.
- TANENBAUM, A. S.; AUSTIN, T. *Organização estruturada de computadores*. 6a edn.. Pearson Prentice Hall, 2013.
- TENNENHOUSE, D. Proactive computing. *Communications of the ACM*, vol. 43, pp. 43–50, 2000.
- TEXAS INSTRUMENTS Tms320c64x dsp library: Programmer's reference. 2003a.
Disponível em: <http://www.ti.com/lit/ug/spru565b/spru565b.pdf>
- TEXAS INSTRUMENTS Tms320c64x image/video processing library. 2003b.
Disponível em: <http://www.ti.com/lit/ug/spru023b/spru023b.pdf>
- TOUATI, S.; BARTHOUD, D. On the decidability of phase ordering problem in optimizing compilation. In: *Proceedings of the 3rd Conf. on Computing Frontiers (CF'06)*, 2006, pp. 147–156.
- VAN DER LOO, M. P. J. The stringdist package for approximate string matching. *The R Journal*, vol. 6, pp. 111–122, 2014.
Disponível em: <http://CRAN.R-project.org/package=stringdist>
- VERILOG, C. Automating circuit design. 2014.
Disponível em: <http://www.c-to-verilog.com/>
- WEHNER, S. Analyzing worms and network traffic using compression. *J. of Computer Security*, vol. 15, no. 3, pp. 303–320, 2007.
- WHEELER, T.; KECECIOGLU, J. Multiple alignment by aligning alignments. In: *Proceedings of the 15th ISCB Conf. on Intelligent Systems for Molecular Biology, Bioinformatics (FCCM'13)*, 2007, pp. i559–i568.
- WILLCOCK, J. J.; LUMSDAINE, A.; QUINLAN, D. J. Reusable, generic program analyses and transformations. In: *Proceedings of the 8th Int. Conf. on Generative Programming and Component Engineering (GPCE'09)*, 2009, pp. 5–14.
- XILINX Microblaze processor reference guide (embedded development kit edk 14.1). 2012.
Disponível em: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf
- ZITZLER, E.; LAUMANN, M.; THIELE, L. *Spea2: Improving the strength pareto evolutionary algorithm*. Tech. Rep. TR-200, Swiss Federal Institute of Technology - Computer Engineering and Networks Lab., Cambridge, MA, 2001.