UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EWERTON DANIEL DE LIMA

Soluções para o Problema da Seleção de Otimizações

Maringá
2013

EWERTON DANIEL DE LIMA

Soluções para o Problema da Seleção de Otimizações

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Anderson Faustino da Silva

Maringá
2013

FOLHA DE APROVAÇÃO

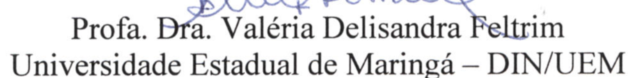
EWERTON DANIEL DE LIMA

Soluções para o problema da seleção de otimizações

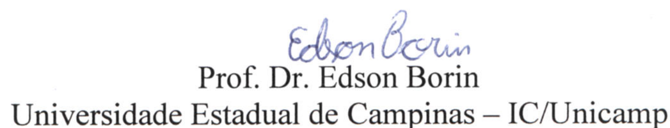
Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação pela Banca Examinadora composta pelos membros:

BANCA EXAMINADORA

Prof. Dr. Anderson Faustino da Silva
Universidade Estadual de Maringá – DIN/UEM



Profa. Dra. Valéria Delisandra Feltrim
Universidade Estadual de Maringá – DIN/UEM



Prof. Dr. Edson Borin
Universidade Estadual de Campinas – IC/Unicamp

Aprovada em: 19 de novembro de 2013.
Local da defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

Lima, Ewerton Daniel de
L732s Soluções para o problema da seleção de otimizações / Ewerton Daniel de Lima. -- Maringá, 2013.
96 f. : il., color., figs., tabs.

Orientador: Prof. Dr. Anderson Faustino da Silva.
Dissertação (mestrado) - Universidade Estadual de Maringá, Centro de Tecnologia, Departamento de Informática, Programa de Pós-Graduação em Ciência da Computação, 2013.

1. Compiladores. 2. Otimizações. 3. Problema de seleção de otimizações. 4. Aprendizagem de máquina. 5. VNS. 6. Eliminação interativa. 7. Algorítimos probabilisticos. I. Silva, Anderson Faustino da, orient. II. Universidade Estadual de Maringá. Centro de Tecnologia. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 21.ed. 005.453

AHS

Soluções para o Problema da Seleção de Otimizações

RESUMO

Códigos gerados por compiladores podem não ter a melhor qualidade possível, devido a dificuldade de obter a sequência de instruções ótima em meio a inúmeras possibilidades. Desenvolvedores de compiladores procuraram melhorar a qualidade do código gerado mediante a implementação de inúmeras otimizações. A aplicação de otimizações, porém, pode prejudicar a qualidade do código, se mal utilizada. Dentre dezenas de otimizações geralmente providas por um compilador é um desafio, até mesmo para o mais experiente programador, saber quais gerarão o melhor código alvo para determinado código fonte. Nesse contexto, o desenvolvimento de seletores de otimizações é um desafio nos dias atuais. Abordagens para a implementação desses seletores são encontradas na literatura e envolvem o uso de buscas aleatórias, exaustivas e heurísticas, algoritmos genéticos e aprendizagem de máquina. Tendo em vista a problemática da seleção automática de otimizações, o presente trabalho apresenta quatro novas abordagens para seleção de bons conjuntos de otimizações. A experimentação das abordagens aqui propostas mostrou que estas possibilitam um ganho de desempenho significativo em comparação a outras propostas encontradas na literatura, sugerindo sua ampla aplicabilidade em contextos diversificados.

Palavras-chave: compiladores, otimizações, problema de seleção de otimizações, aprendizagem de máquina, VNS, eliminação iterativa, algoritmos probabilísticos

Solutions for Optimization Selection Problem

ABSTRACT

Compiler generated codes may not have the best quality possible because it is difficult to obtain the optimal sequence of instructions as it has endless possibilities. Compilers developers tried to improve code quality implementing some optimizations. However, when it is not used correctly, the application of optimizations may impair code quality. Among dozens of optimizations usually provided by a compiler, it is a challenge to know which ones will generate a better target code for a specific source code, even for the most experienced programmer. In this context, development of automated optimizations selectors is a challenge today. Approaches to the implementation of these selectors are found in the literature and include use of random, exhaustive and heuristics searches, genetic algorithms and machine learning. In view of the optimizations automatic selection problematic, this work presents four new approaches for selection of good optimizations sets. The experimental evaluation of this approaches showed that they make it possible a significant performance gain compared to approaches found in literature and suggests a wide applicability in various contexts.

Keywords: compilers, optimizations, optimizations selection problem, machine learning, VNS, iterative elimination, probabilistic algorithms

LISTA DE FIGURAS

2.1 Aplicação da Substituição Escalar de Agregados 16
2.2 Aplicação da Propagação de Cópia 17
2.3 Aplicação da Propagação de Constante 17
2.4 Código em que pode ser aplicada a Numeração de Valores 18
2.5 Aplicação da Eliminação de Subexpressão Comum 18
2.6 Aplicação da Movimentação de Invariante de Laço 18
2.7 Aplicação da otimização *Code Hoisting*. Adaptado de Muchnick (1997) 19
2.8 Aplicação da Eliminação de Verificação Desnecessária de Faixa de Valores 20
2.9 Processo de otimização de um determinado código mediante a aplicação das otimizações X, Y e Z 21

3.1 *Speedups*, em relação à O0, da aplicação da técnica estatística de Haneda et al. (2005) 28

4.1 Arquitetura do *CBR-Selector* 52

5.1 *Speedups* alcançados pela metaheurística VNS 62
5.2 Número de avaliações para as soluções encontradas pela VNS . . . 63
5.3 *Speedups* e número de avaliações da VNS 63
5.4 *Speedups* alcançados pelo PBE 64
5.5 Probabilidades e *speedups* do PBE 66
5.6 *Speedups* alcançados pelo IPBE 68
5.7 Probabilidades e *speedups* do IPBE 69
5.8 *Speedups* da avaliação Treino e Teste 75
5.9 *Speedups* da avaliação *Leave-One-Out Cross Validation* 76
5.10 *Speedups* alcançados pelos algoritmos BE, IE e CE 79
5.11 Número de avaliações para as soluções encontradas pelo BE, IE e CE 80
5.12 *Speedups* e número de avaliações do BE, IE e CE 80
5.13 Comparação ganho de desempenho *versus* número de avaliações para todas abordagens experimentadas 82

LISTA DE TABELAS

3.1 Comparação entre as estratégias utilizadas nos trabalhos relacionados com as estratégias propostas 40

5.1 Ganho de desempenho médio para cada número de tentativas do PBE 66

5.2 Ganho de desempenho médio para cada número de tentativas do IPBE 70

5.3 Otimizações da LLVM utilizadas pelo *CBR-Selector* 72

5.4 Número de conjuntos com desempenho melhor que `-O3` para cada programa do espaço exploratório 73

5.5 Quadro comparativo dos resultados das abordagens avaliadas . . . 83

LISTA DE SIGLAS E ABREVIATURAS

AG: Algoritmo Genético
AM: Aprendizagem de Máquina
BE: *Batch Elimination*
BEIECE: *Batch, Iterative and Combined Eliminations*
CE: *Combined Elimination*
CEE: Construtor de Espaço Exploratório
CRC: *Cyclic Redundancy Code*
EE: Espaço Exploratório
GCC: *GNU Compiler Collection*
IE: *Iterative Elimination*
IPBE: *Improved Probabilistic Batch Elimination*
JIT: *Just-In-Time*
JVM: *Java Virtual Machine*
LLVM: *Low Level Virtual Machine*
LOO: *Leave-One-Out*
NEAT: *Neuro-Evolution for Augmenting Topologies*
NPB: NASA *Parallel Benchmark*
OPN: Otimização Possivelmente Nociva
PAPI: *Performance Application Programming Interface*
PBE: *Probabilistic Batch Elimination*
PC: *Performance Counter*
PSO: Problema de Seleção de Otimizações
RBC: Raciocínio Baseado em Casos
RNA: Rede Neural Artificial
SPARC: *Scalable Processor Architecture*
SPEC: *Standard Performance Evaluation Corporation*
SVM: *Support Vector Machine*
TT: Treino e Teste
VNS: *Variable Neighborhood Search*

SUMÁRIO

1 Introdução **11**

2 O Problema da Seleção de Otimizações **15**

2.1 Compiladores e Otimizações 15

2.2 Seleção de Otimizações 21

2.2.1 A Ordem de Aplicação das Otimizações 23

2.3 Considerações Gerais 24

3 Trabalhos Relacionados **25**

3.1 Abordagens com Busca Exaustiva 25

3.2 Abordagem com Técnica Estatística 26

3.3 Abordagens com Eliminação Iterativa 29

3.4 Abordagens com Algoritmos Genéticos 30

3.5 Abordagens com Aprendizagem de Máquina 32

3.6 Abordagens Híbridas 36

3.7 Considerações Gerais 38

4 Abordagens Propostas para Mitigar o Problema da Seleção de Otimizações **41**

4.1 *Variable Neighborhood Search* 41

4.1.1 Operadores de Vizinhança 42

4.1.2 Rotina de Perturbação 43

4.1.3 Busca Local 43

4.1.4 Algoritmo VNS para o PSO 46

4.2 *Probabilistic Batch Elimination* 47

4.3 *Improved Probabilistic Batch Elimination* 49

4.4 *CBR-Selector* 51

4.4.1 Construtor do Espaço Exploratório 52

4.4.2 Seletor de Conjuntos 54

4.4.3 Avaliador de Conjuntos 55

4.5 Considerações Gerais 57

5 Avaliação Experimental **58**

5.1 Metodologia 58

5.2 *Variable Neighborhood Search* 61

5.2.1 *Speedups* 61
5.2.2 Número de Avaliações 62
5.3 *Probabilistic Batch Elimination* 64
5.3.1 *Speedups* 64
5.3.2 Probabilidades e *Speedups* 65
5.3.3 Número de Avaliações 66
5.4 *Improved Probabilistic Batch Elimination* 67
5.4.1 *Speedups* 67
5.4.2 Probabilidades e *Speedups* 68
5.4.3 Número de Avaliações 69
5.5 *CBR-Selector* 70
5.5.1 Configurações Específicas 71
5.5.2 *Speedups* 73
5.5.3 Número de Avaliações 77
5.6 *Batch, Iterative and Combined Eliminations* 78
5.6.1 *Speedups* 78
5.6.2 Número de Avaliações 80
5.7 Resumo das Abordagens Avaliadas 81
5.8 Considerações Gerais 84

6 Conclusões e Trabalhos Futuros **85**
6.1 Trabalhos Futuros 86
6.2 Considerações Finais 87

Referências **89**

1

Introdução

Compilador é uma ferramenta que geralmente traduz código de linguagens de alto nível (Scott, 2009; Sebesta, 2009), legíveis a seres humanos, em linguagem de montagem (*Assembly*), a qual pode ser facilmente transformada em código binário por um utilitário montador (*Assembler*) (Blum, 2005). Essa tradução não é trivial e é feita em etapas, que podem variar de acordo com a implementação do compilador (Aho et al., 2006). No entanto, independente da quantidade de etapas, estas são divididas em fases de análise e fases de tradução, as quais se comportam como segue:

Fases de análise realizam verificações a procura de possíveis erros no código fonte. Em geral, os compiladores implementam as seguintes análises: léxica, sintática e semântica.

Fases de tradução realizam transformações no código fonte até que este se torne em código *assembly*. Nessa etapa é possível implementar as seguintes etapas: geração de uma representação intermediária, manipulações de blocos básicos, aplicação de otimizações, seleção de instruções para a máquina alvo, alocação de registradores e emissão de código executável.

O que ocorre no processo de compilação, em geral, é que um pequeno trecho de código fonte é transformado em uma sequência maior de instruções escritas em linguagem *Assembly*. Assim, para um determinado código fonte é possível gerar várias sequências distintas, as quais produzem o mesmo resultado semântico. A escolha de tal sequência implica diretamente nos seguintes fatores: tamanho do código alvo, velocidade de execução, consumo de energia, consumo de memória, dentre outros fatores.

Compiladores que aplicam otimizações, denominados compiladores otimizadores, fornecem geralmente dezenas de otimizações. Dentre elas é possível escolher quais serão aplicadas, pois nem toda otimização modificará o código de maneira benéfica, podendo em alguns casos até ocasionar perda de desempenho.

Existem pelo menos três questões fundamentais que afetarão a qualidade do código ao se aplicar otimizações, a saber:

1. **Quais otimizações aplicar?** Aplicar todas as otimizações possíveis não significa necessariamente obter o melhor código, dado que as características do código fonte podem não se adequar às características específicas de cada otimização.

2. **Em qual ordem as otimizações devem ser aplicadas?** Outro fator responsável pela qualidade do código final é a ordem em que as otimizações são aplicadas. Isso devido ao fato de uma otimização poder neutralizar, melhorar ou até mesmo piorar os resultados de suas antecessoras e/ou sucessoras.

3. **Quais parâmetros devem nortear a aplicação de cada otimização?** Algumas otimizações são parametrizáveis, o que significa que a forma como elas são aplicadas varia de acordo com uma determinada configuração. A boa escolha dessa configuração normalmente é dependente do programa fonte, pois o comportamento da otimização é sensível às características do código.

Neste contexto, o Problema da Seleção de Otimizações (PSO) é: *escolher o melhor conjunto de otimizações e seus respectivos parâmetros e configuração de ordem para um código X, tal que X seja ótimo para um dado objetivo.*

Uma forma de garantir o resultado ótimo é avaliar todos possíveis conjuntos de otimizações, levando em consideração a ordem e seus parâmetros. No entanto, não é difícil perceber que tal metodologia é inviável em termos de tempo de processamento. Considerando tão somente os conjuntos de otimizações possíveis, sem considerar a ordem e os parâmetros, a quantidade de avaliações para um determinado código seria de 2^n, onde n é o número de otimizações possíveis de serem aplicadas. Assim, um compilador que oferece a possibilidade de aplicação de 61 otimizações diferentes, como é o caso do Clang (Lattner, 2002), necessitaria de:

$$2^n = 2^{61} = 2.305.843.009.213.693.952 \text{ avaliações}$$

Isso significa que para garantir o melhor subconjunto de otimizações fazendo a experimentação de todas as combinações possíveis de um programa que compila e executa em 1 segundo seriam necessários cerca de 73 bilhões de anos de processamento.

Na tentativa de procurar soluções viáveis para o PSO, várias abordagens foram propostas na literatura. Dentre essas abordagens estão: buscas exaustivas (Foleiss et al., 2011a,b; Kulkarni et al., 2006), técnica estatística (Haneda et al., 2005), eliminação iterativa (Pan e Eigenmann, 2006), algoritmos genéticos (Almagor et al., 2004; Cooper et al., 1999; Leather et al., 2009), aprendizagem de máquina (Cavazos et al., 2007; Park et al., 2011) e também abordagens que combinam duas ou mais dessas técnicas (Lau et al., 2006; Purini e Jain, 2013).

Segundo Kulkarni et al. (2005), em alguns contextos, a abordagem de busca exaustiva pode ser utilizada. O trabalho de Foleiss et al. (2011b) utilizou busca exaustiva em algumas classes específicas de otimização, sendo a busca feita por classes e não por cada otimização isoladamente. Essa restrição permitiu que fosse viável a busca exaustiva no problema apresentado. No entanto, esse tipo de abordagem está restrito a contextos bem específicos.

Técnicas estatísticas foram utilizadas no trabalho de Haneda et al. (2005), onde o teste de hipótese de Mann-Whitney (Tan et al., 2005) serviu para verificar se determinada otimização afeta ou não o código gerado, permitindo assim decidir se é viável selecioná-la ou não.

O trabalho de Pan e Eigenmann (2006) utilizou eliminação iterativa para selecionar bons conjuntos de otimizações. O objetivo foi investigar quais otimizações prejudicavam a qualidade para determinado código e então retirá-las do conjunto padrão.

Abordagens genéticas foram utilizadas tanto para busca de bons conjuntos de otimização em termos de *speedup* e tamanho de código (Almagor et al., 2004; Cooper et al., 1999) quanto para construções de heurísticas a serem utilizadas em otimizações específicas como, por exemplo, para decidir o melhor nível de aplicação da otimização *loop unroll* (Leather et al., 2009).

Alguns trabalhos como os de Cavazos et al. (2007) e Park et al. (2011) utilizam uma abordagem aleatória para construir um conjunto de treino ao qual serão aplicados algoritmos de aprendizagem de máquina capazes de prever bons conjuntos de otimizações dadas as características de um programa.

O trabalho de Lau et al. (2006) combinou aleatoriedade e técnicas estatísticas para determinar quais otimizações são boas pra uma determinada região de código no contexto das máquinas virtuais (Smith e Nair, 2005) e compilação JIT.

Uma abordagem mais recente para o PSO (Purini e Jain, 2013) combinou pelo menos quatro técnicas diferentes para seleção de otimizações, a saber: busca aleatória, algoritmos genéticos, aprendizagem de máquina e eliminação iterativa.

O presente trabalho propõe quatro novas técnicas para seleção de otimizações, as quais são diferentes entre si, tanto em características quanto em propósitos. Uma delas utiliza a clássica metaheurística *Variable Neighborhood Search* (VNS), duas combinam as técnicas de eliminação iterativa e algoritmos probabilísticos e, por fim, a última utiliza raciocínio baseado em casos, uma técnica de aprendizagem de máquina. Tais técnicas visam minimizar o problema da escolha de otimizações que não são parametrizáveis, sendo o problema de ordem das otimizações abordados de forma indireta.

As principais contribuições deste trabalho são:

1. Implementação de quatro algoritmos de seleção de otimizações;
2. Implementação de operadores de busca local adequados ao PSO, os quais são utilizados pela metaheurística VNS;
3. Modificação de algoritmos clássicos de seleção de otimizações, originando estratégias mais eficientes;
4. Evidência de que não é necessário combinar muitas técnicas para alcançar *speedups* significativos;
5. Evidência de que é possível selecionar uma boa sequência de otimizações com poucas avaliações.

Dentre as quatro estratégias propostas, houve *speedup* médio de até 16,9% no conjunto de *benchmarks MiBench* (Guthaus et al., 2001), porém com um alto número de avaliações. A abordagem que utilizou raciocínio baseado em casos alcançou a um ganho de desempenho médio de 10,10% em uma experimentação *leave-one-out* com 21 programas do *MiBench* e até 8,8% em uma experimentação treino e teste em 25 programas de diversos conjuntos de *benchmarks*, sendo necessárias apenas 20 avaliações para que esses resultados fossem alcançados.

O texto segue com a seguinte organização: o Capítulo 2 detalha o PSO descrevendo inicialmente o funcionamento de algumas otimizações e em seguida as dificuldades em selecionar bons conjuntos das mesmas para um determinado programa. O Capítulo 3 apresenta detalhes dos trabalhos apresentados na literatura para mitigação do problema. O Capítulo 4 descreve os quatro mecanismos de seleção de otimizações propostos, sendo a avaliação experimental deles apresentada no Capítulo 5, juntamente com a experimentação do trabalho de Pan e Eigenmann (2006), para fins comparativos. Por fim, o Capítulo 6 apresenta as conclusões inferidas a partir deste estudo e aponta trabalhos que poderão ser realizados futuramente em decorrência das mesmas.

2

O Problema da Seleção de Otimizações

Este capítulo descreve a etapa de aplicação de otimizações e aborda as principais questões do Problema de Seleção de Otimizações (PSO).

2.1 Compiladores e Otimizações

Compiladores modernos aplicam modificações no código, com o objetivo de melhorar o seu desempenho. Essas modificações são denominadas otimizações ou transformações. Um exemplo clássico de otimização é *inline* (Muchnick, 1997), que substitui uma chamada de função pelo seu próprio corpo. Assim, quando o código for executado, no momento em que chegar aos pontos de chamadas de função não haverá a necessidade da manutenção da pilha. Em geral o uso dessa otimização resulta em uma execução mais rápida (Miller et al., 2010).

Compiladores modernos como Clang (Lattner, 2002) e GCC (Stallman e Developer-Community, 2009) disponibilizam ao usuário dezenas de otimizações e fornecem a opção da escolha de quais otimizações aplicar. No entanto, o mais comum é o uso de conjuntos de otimizações pré-definidos que são específicos para um propósito como, por exemplo, reduzir o tempo de execução do código gerado.

Os projetistas de compiladores procuram agrupar as otimizações de acordo com suas peculiaridades. Segundo Muchnick (1997), as otimizações podem ser agrupadas nas seguintes classes:

- Otimizações fundamentais;

- Otimizações de eliminação de redundância;
- Otimizações de laços; e
- Otimizações de procedimento.

As otimizações fundamentais envolvem todo conjunto de otimizações básicas e clássicas, tais como: avaliação de expressão-constante, substituição escalar de agregados, simplificação algébrica e reassociação, propagação de cópia e propagação de constante. As três primeiras não requerem uma análise de fluxo de dados (Muchnick, 1997) para serem executadas. Já as três últimas a requerem. Uma descrição do funcionamento de cada otimização pode ser vista a seguir.

Avaliação de expressão-constante Em tempo de compilação as expressões cujos operandos são constantes são substituídas pelo valor resultante. Por exemplo, a expressão $2 * 3 + 1$ poderia ser substituída pela constante 7, evitando as operações de multiplicação e adição da expressão em tempo de execução.

Substituição escalar de agregados Essa otimização estima quais variáveis de um agregado heterogêneo (como as *structs* em C) (Sebesta, 2009) mantêm um valor constante e substitui sua referência por um temporário. Isso torna possível uma posterior aplicação da otimização de eliminação de código morto. Um exemplo de aplicação dessa otimização pode ser visto na Figura 2.1

```
typedef struct client {
        char name[30];
        int code;
} CLIENT;

int credit(CLIENT *c) {
        switch (c->code) {
                case 0: return 300; break;
                case 1: return 1000; break;
                case 2: return 2000; break
        }
}

main() {
        CLIENT client;
        client.code = 0;
        printf("%d", credit(&client));
}
```

(a) Antes

```
main() {
int t1 = 0;
int t2;

        switch (t1) {
                case 0: t2 = 300; break;
                case 1: t2 = 1000; break;
                case 2: t2 = 2000; break
        }

printf("%d", t2);
}
```

(b) Depois

Figura 2.1: Aplicação da Substituição Escalar de Agregados

Simplificação algébrica e reassociação Essa otimização utiliza princípios de equivalência matemática para simplificar expressões algébricas. Um exemplo dessa otimização pode ser vista na expressão $(i - j) + (i - j) + (i - j) + (i - j)$, que poderia ser substituída por $4 * i - 4 * j$, após a aplicação dessa otimização.

Propagação de cópia Para uma dada atribuição $x = y$, a propagação de cópia substitui todas as ocorrências seguintes de x por y. A Figura 2.2 apresenta um exemplo de aplicação desta otimização.

```
b = a;
c = 4 * b;
d = a + b;

if b > 10 return;
```

(a) Antes

```
b = a;
c = 4 * a;
d = a + a;

if a > 10 return;
```

(b) Depois

Figura 2.2: Aplicação da Propagação de Cópia

Propagação de constante Para uma dada atribuição $x = c$, em que x é uma variável e c é uma constante, essa otimização substitui todas ocorrências seguintes da variável x pela constante c. A Figura 2.3 apresenta um exemplo.

```
b = 10;
c = 4 * b;
d = a + b;
```

(a) Antes

```
b = 10;
c = 4 * 10;
d = a + 10;
```

(b) Depois

Figura 2.3: Aplicação da Propagação de Constante

Nas otimizações de eliminação de redundância, o compilador analisa o código com o objetivo de encontrar computações repetidas e então as elimina. Essas otimizações podem favorecer tanto a redução do tamanho de código quanto a redução de tempo de processamento. Estão nessa classe as seguintes otimizações: numeração de valores, eliminação de sub expressão comum, movimentação de código invariante de laço e *code hoisting*. A seguir é fornecida uma breve descrição de cada otimização dessa classe.

Numeração de valores É um dos métodos que verifica duas computações equivalentes e elimina uma delas. Na Figura 2.4, o método de numeração de valores detecta que as variáveis j e l possuem o mesmo resultado e elimina a redundância.

Eliminação de sub expressão comum Quando uma expressão é repetida no código, essa otimização armazena o resultado da primeira computação e o substitui nas

```
j = i + 1;
k = i;
l = k + 1
```

Figura 2.4: Código em que pode ser aplicada a Numeração de Valores

expressões seguintes. Por exemplo, ao aplicar essa otimização no código apresentado na Figura 2.5(a), a primeira expressão $a + 2$ armazena seu resultado em um temporário e o substitui nas expressões com computação equivalente seguintes, resultando no código da Figura 2.5(b).

```
b = a + 2;
c = d + f;
d = a + 2;
```

(a) Antes

```
t1 = a + 2;
b = t1;
c = d + f;
d = t1;
```

(b) Depois

Figura 2.5: Aplicação da Eliminação de Subexpressão Comum

Movimentação de código invariante de laço Essa otimização procura por computações dentro de laços que sempre produzem o mesmo resultado e remove-as do corpo de laço. Assim, um código como o da Figura 2.6(a) poderia ser transformado no da Figura 2.6(b).

```
for (int i = 1; i <= 100; i++) {
    a = i * 100 * (n + 2);
}
```

(a) Antes

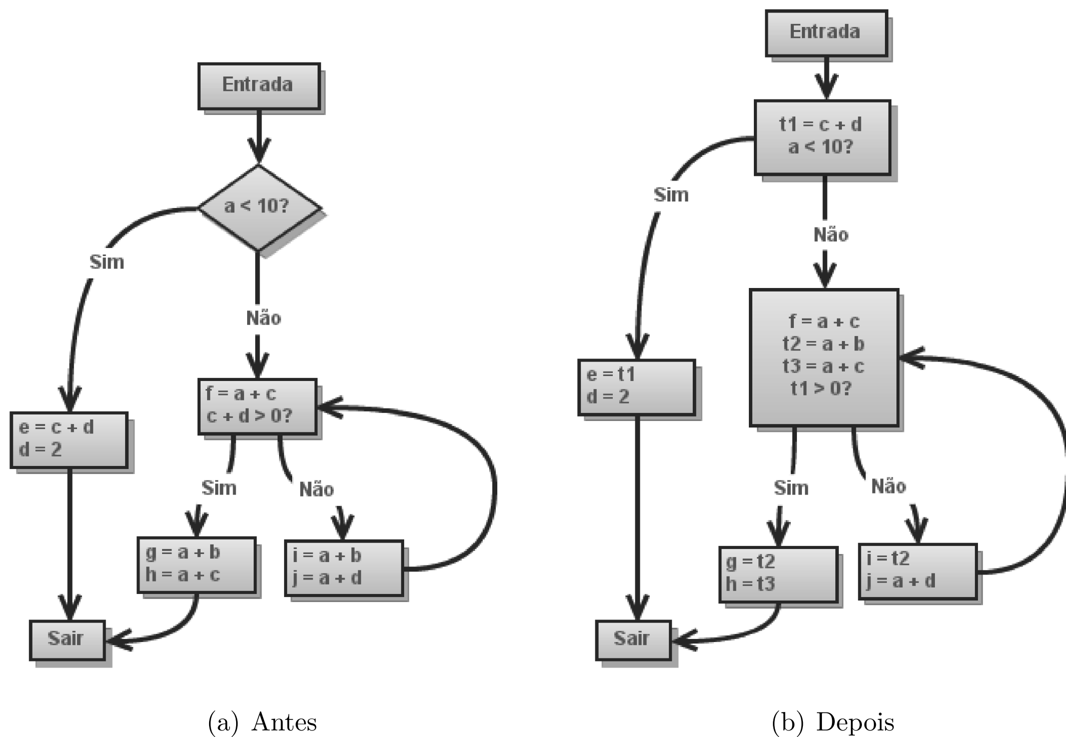
```
t1 = 100 * (n + 2);
for (int i = 1; i <= 100; i++) {
    a = i * t1;
}
```

(b) Depois

Figura 2.6: Aplicação da Movimentação de Invariante de Laço

Code hoisting Também é conhecida pelo termo unificação. Procura "adiantar" a execução de algumas computações que estão depois de uma expressão condicional, com o objetivo de postergar o máximo possível a decisão. Pode afetar o tempo de execução tanto positiva quanto negativamente. Um exemplo de aplicação da otimização *code hoisting* pode ser visto na Figura 2.7.

As otimizações de laços são otimizações que são aplicáveis somente a laços de repetição ou têm melhores resultados quando aplicadas em laços de repetição. Dentre as otimizações dessa classe estão: otimizações de variáveis de indução e eliminação de verificação desnecessária de faixa de valores, que são descritas e exemplificadas a seguir.



(a) Antes (b) Depois

Figura 2.7: Aplicação da otimização *Code Hoisting*. Adaptado de Muchnick (1997)

Otimizações de variáveis de indução Essa é uma subclasse de otimizações que exploram as variáveis de indução, que são variáveis que recebem valores sucessivamente na forma de uma progressão aritmética. `Redução de força` é um exemplo de otimização nessa subclasse. Ela procura por operações aritméticas mais "caras" (como multiplicação e divisão) e tenta substituí-las por operações mais simples (como adição e subtração). Assim, a sequência 0, 3, 6, 9, 12, gerada por uma operação $3 * i$ dentro de um laço, poderia ser substituída por $i + 3$, reduzindo o custo computacional.

Eliminação de verificação desnecessária de faixa de valores Há linguagens que permitem definir tipos inteiros restritos a uma faixa de valores, como Java (Würthinger et al., 2007) e Fortran (Nguyen e Irigoin, 2005). Assim, quando é feita uma atribuição de um valor fora dessa faixa, uma exceção deve ser lançada em tempo de execução. No entanto, em laços em que se percorre uma faixa de valores não é necessário fazer tal verificação a cada iteração. Portanto, essa otimização faz as modificações necessárias para evitar tais repetições, como é mostrado na Figura 2.8.

A última classe de otimizações aqui mencionada é a denominada "otimizações de procedimento". Algumas otimizações dessa classe são: eliminação de recursão em cauda

```
i = init
L1:
        if (i < low) exit(1);
        if (i > high) exit(1);
        //uso do indice i
        i = i + 1;
        if (i <= fin) goto L1
```

(a) Antes

```
if (init < low) exit(1);
t1 = min(fin,high);
i = init;
L1:
        //uso do indice i
        i = i + 1;
        if (i <= t1) goto L1
        if (i <= fin) exit(1);
```

(b) Depois

Figura 2.8: Aplicação da Eliminação de Verificação Desnecessária de Faixa de Valores

e otimização de chamada em cauda, *inline*, otimização de rotina-folha e *shrink wrapping* (que generaliza a otimização de rotina-folha). A seguir são descritas e exemplificadas essas otimizações.

Eliminação de chamada em cauda ou recursão em cauda Se o procedimento $G()$ chama o procedimento $F()$ e, após a chamada a única coisa que se faz é retorná-la, então essa é considerada uma "chamada em cauda", ou seja, uma chamada é considerada em cauda se é a última coisa executada dentro do escopo de uma função. Isso também é válido para chamadas recursivas (quando $G() = F()$). Assim, a eliminação de chamada/recursão em cauda transforma as chamadas em cauda em laços de repetição, evitando os custos computacionais de manutenção das chamadas de função.

Inline substitui chamadas de função pelo próprio corpo da função, para reduzir custos computacionais de tratamento das chamadas de função.

Otimização de rotina-folha Um procedimento é chamado rotina-folha se é uma folha no grafo de chamadas de procedimento. Essa otimização tira vantagem desse fato para tentar simplificar a forma como os parâmetros são passados para ele e também remover o máximo possível das partes que fazem chamadas dentro do código. É importante observar que um alto percentual de procedimentos são rotina-folha, que é a motivação para aplicação de tal otimização.

Após o usuário selecionar cada uma das otimizações ou mesmo o nível que deseja aplicar, o compilador aplica-as em uma determinada ordem. Em alguns compiladores essa ordem é fixa e baseia-se em alguns estudos que sugerem que determinadas classes de otimizações devem ser aplicadas antes de outras para se obter melhores resultados (Muchnick, 1997). Assim, há implicações no código final, dependendo da ordem em que são aplicadas as otimizações.

Por exemplo, se as otimizações X, Y e Z são aplicadas no código A, então o código A será transformado em A_x, depois em A_{xy} e, por fim, em A_{xyz}, como mostra a Figura

2.9. O código A, após a aplicação das otimizações X, Y e Z, nesta ordem, se transforma no código A_{xyz}, que não é o mesmo que o A_{yxz} ou A_{zxy}. Portanto, o código final gerado dependerá também da ordem em que são aplicadas as otimizações.

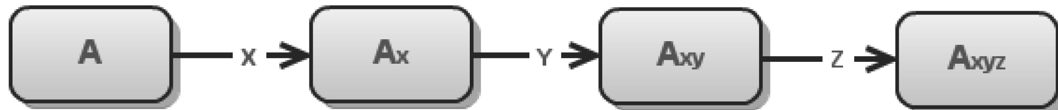


Figura 2.9: Processo de otimização de um determinado código mediante a aplicação das otimizações X, Y e Z

A ordem de aplicação das otimizações sem dúvida deve ser considerada no momento da seleção de otimizações, pois a mudança de ordem gera resultados diversos. A seção seguinte aborda a problemática da seleção de otimizações considerando, inclusive, a questão da ordem de aplicação. Neste ponto, é importante mencionar que nem todos compiladores permitem ao usuário decidir uma ordem absoluta na qual serão aplicadas as otimizações (Cavazos et al., 2007).

2.2 Seleção de Otimizações

Com as informações citadas na seção anterior é possível perceber que as otimizações visam melhorar um código gerado por um compilador. No entanto, nem sempre isso acontece devido ao fato de as características de determinado código nem sempre favorecerem a aplicação de determinada otimização. Por exemplo, a otimização *inline* geralmente melhora a velocidade de execução do programa, já que a replicação do corpo das funções pode favorecer o princípio da localidade de arquitetura de computadores (Hennessy e Patterson, 2006). No entanto, para alguns programas a *inline* pode aumentar o código consideravelmente de modo a haver a necessidade de mais buscas de instruções em memória, o que deve acarretar perda de desempenho.

Considerando ainda esta otimização, é possível também observar que se não houver chamadas de função no código, ele não sofrerá modificações e, consequentemente, não será otimizado. Isso pode significar um custo adicional de processamento desnecessário durante a compilação.

Essas observações sugerem que o resultado de uma otimização depende diretamente das características do programa ao qual está sendo aplicada, ou seja, nem sempre é

viável aplicar determinada otimização. Tudo dependerá das características estruturais do programa, como também do seu comportamento.

Considerando tais dificuldades na seleção de otimizações, o PSO pode considerar três situações distintas, a saber:

Seleção de um conjunto Neste caso o problema limita-se a encontrar qual o melhor conjunto de otimizações, considerando que há uma ordem de aplicação pré-fixada, que não pode ser modificada;

Seleção de uma sequência Neste caso o problema considera tanto quais são as melhores otimizações para o código fonte, bem como qual é a ordem em que elas devem ser aplicadas. É feita a seleção de uma sequência de modo que cada otimização ocorra no máximo uma vez;

Seleção de uma sequência com repetições Este último caso considera, além da ordem em que as otimizações são aplicadas, a possibilidade de repetir a aplicação de otimizações.

Para todos os casos acima, uma abordagem exaustiva é impraticável, dada a necessidade de compilar e avaliar o código fonte para cada possibilidade de sequência.

No primeiro caso, como devem ser analisados todos os possíveis subconjuntos de um conjunto T de otimizações, o número de avaliações que devem ser feitas é de $\#\mathcal{P}(T) = 2^n$, em que $\mathcal{P}(T)$ é o conjunto das partes do conjunto T e n é o tamanho de T (Gerônimo e Franco, 2008). A expressão 2^n indica claramente que o número de avaliações a serem feitas cresce exponencialmente em função da quantidade de otimizações disponíveis para serem aplicadas.

No segundo caso, o número de avaliações é ainda maior, já que a ordem é considerada. Como para cada subconjunto de T todas possíveis ordens devem ser experimentadas, o número total de avaliações é de $A_0^n + A_1^n + A_2^n + A_3^n + ... + A_i^n + ... + A_{n-2}^n + A_{n-1}^n + A_n^n$, onde A_i^n é o número de possíveis arranjos de n elementos tomados i a i (Gerônimo e Franco, 2008).

Por fim, no terceiro caso, como devem ser exploradas todas as possibilidades, incluindo a repetição de uma ou mais otimizações, o espaço de busca é infinito. É importante mencionar que apesar de esse espaço ser teoricamente infinito, há na prática uma limitação na repetição de aplicação de otimizações nos compiladores.

Assim, no primeiro caso há um problema com espaço de busca que cresce exponencialmente em função do número de otimizações do compilador, no segundo há um espaço de busca com crescimento fatorial e no terceiro há um espaço de busca infinito.

Independente da versão do problema, o espaço de busca é muito grande para ser explorado exaustivamente. Assim, estratégias eficientes de exploração desses espaços devem ser propostas para reduzir o tempo de encontrar soluções para o PSO.

2.2.1 A Ordem de Aplicação das Otimizações

Como mencionado anteriormente, o código resultante das otimizações também depende da ordem em que elas são aplicadas. O objetivo desta subseção é apresentar o problema da ordem de aplicação das otimizações, mediante um exemplo que considera as seguintes otimizações:

Reassociação de expressões a qual reassocia expressões para facilitar a eliminação de expressões constante. Por exemplo, a expressão $4 + (x + 5)$ é transformada em $x + (4 + 5)$;

Propagação e eliminação de expressões constante que transforma instruções de operações simples entre constantes, como add $i32, 1, 2$ em mov $i32, 3$, tornando desnecessária a operação de adição;

Eliminação de instruções mortas que remove instruções que não são úteis na execução do programa.

Considerando a expressão: $4 + (x + 5)$ e as três otimizações citadas anteriormente, aplicadas na ordem em que foram citadas, o código já passado pela reassociação de expressões:

```
add DEST1, 4, 5
add DEST2, DEST1, x
```

Será transformado pela propagação e eliminação de expressões constante em:

```
add DEST1, 4, 5
add DEST2, 9, x
```

A eliminação de instruções mortas então eliminará a primeira instrução, pois seu operando de destino “DEST1” nunca é utilizado. Portanto o código final gerado será:

```
add DEST2, 9, x
```

Se as mesmas três otimizações fossem aplicadas na ordem contrária, a expressão $4 + (x + 5)$, poderia ser traduzida como:

```
add DEST1, x, 5
add DEST2, DEST1, 4
```

Iniciando com eliminação de instruções mortas, observa-se que nenhuma eliminação é possível nesse código. A propagação de constantes também não pode ser aplicada. Por fim, a reassociação seria aplicada, transformando a expressão em $x + (4 + 5)$, o que não melhoraria o código, pois as otimizações que se beneficiam com essa modificação já foram aplicadas. Esse exemplo indica que a ordem de aplicação das otimizações é relevante no processo de seleção, aumentando ainda mais a complexidade do PSO.

O processo de seleção de otimizações para cada programa é um desafio. Se a escolha de otimizações for uma tarefa de responsabilidade do programador, este deverá ter conhecimento minucioso das características de seu código, dos efeitos e implicações da aplicação de cada otimização sobre o código e provavelmente conhecimento de como foram implementadas no compilador. Com esses conhecimentos, o usuário teria condições de escolher, mediante árdua tarefa, boas combinações de otimizações para seu código. Dada a complexidade de tal tarefa para muitos programadores e também a não existência de um algoritmo de seleção com solução exata e em tempo polinomial, a seleção automática de um conjunto de boas otimizações é um desafio a ser explorado.

2.3 Considerações Gerais

Este capítulo contextualizou o Problema da Seleção de Otimizações (PSO), apresentando brevemente o funcionamento de algumas otimizações de compiladores e mostrando a dificuldade existente em escolher o melhor conjunto de otimizações para determinado código. Foi ainda exposta a questão da ordem de aplicação das otimizações, que também interfere na qualidade do código final. Para uma compreensão mais ampla do problema, o capítulo seguinte apresenta estratégias encontradas na literatura propostas para mitigação do PSO.

3

Trabalhos Relacionados

Este capítulo apresenta trabalhos desenvolvidos na busca de soluções para o PSO, os quais utilizam diferentes estratégias, a saber: busca exaustiva, técnica estatística, eliminação iterativa, algoritmos genéticos, aprendizagem de máquina e híbridas.

3.1 Abordagens com Busca Exaustiva

Um estudo de aplicação de otimizações para redução de tamanho de código pode ser visto no trabalho de Foleiss et al. (2011a). Restritos às aplicações de redes de sensores sem fio, os autores avaliaram classes de otimizações com o objetivo de analisar o impacto de cada uma no tamanho de código gerado. O total de classes de otimizações aplicadas foi de 12 classes: classes de otimizações de controle de fluxo, *inline*, procedimento, propagações, eliminações, variáveis, otimizações de laços, *crossjumping*, registradores, *peephole*, reordenação e *cleanup*. Além disso, analisaram também o impacto dos diferentes níveis de otimização do AVR-GCC (Fryza, 2007) no tamanho de código. Em um trabalho posterior (Foleiss et al., 2011b), optaram por realizar uma busca exaustiva para selecionar otimizações para tamanho de código. Baseando-se em estudos anteriores (Bungo, 2008; Cavazos et al., 2006; Triantafyllis et al., 2003), foram consideradas apenas otimizações que impactavam significativamente o tamanho do código. Em ambos trabalhos uma busca exaustiva foi factível devido à redução do espaço de busca.

O diferencial do segundo trabalho foi a construção de uma base de conhecimento para a aplicação do algoritmo *Apriori* (Agrawal e Srikant, 1994) e assim inferir bons conjuntos de otimização para reduzir o tamanho dos códigos compilados.

A questão da ordem de aplicação das otimizações é explorada por Kulkarni et al. (2006) também com uma abordagem exaustiva. Esse trabalho indicou que o espaço de busca para encontrar a melhor ordem de aplicação das otimizações não é tão grande se forem aplicadas algumas técnicas de poda juntamente com a busca exaustiva.

A proposta de Kulkarni et al. (2006) apresenta duas técnicas para reduzir o espaço de busca. A primeira explora o fato de que nem sempre uma otimização modifica o código. Há casos em que ela não encontra oportunidades para executar e o código de entrada passa a ser o mesmo na saída. Com o conhecimento de que determinada etapa não modifica o código, é possível eliminar ramos inteiros da árvore de busca.

A segunda técnica de poda se baseia na assertiva de que muitas otimizações em vários níveis produzem instâncias de função que são idênticas àquelas já encontradas em níveis anteriores ou àquelas geradas por sequências anteriores no mesmo nível. Para identificar tais códigos e realizar a poda é aplicado um algoritmo de similaridade rápido baseado em três atributos: o número de instruções, a soma dos *bytes* de todas instruções e o CRC (*cyclic-redundancy code*) de cada código.

Aplicando essas duas técnicas de poda na busca exaustiva, os resultados apresentaram uma grande redução no espaço de busca. Entre os 111 códigos avaliados (111 funções do conjunto de *benchmarks MiBench*), apenas duas não puderam ter seu espaço de busca totalmente enumerado. Em média, o tamanho do espaço de busca ficou em 15^{12}, podendo crescer até 15^{32} no pior caso. As técnicas de poda apresentadas deixaram o espaço de busca para os códigos apresentados com tamanho 15^{12}, que representa uma fração ínfima do espaço de busca total, 15^{32}.

3.2 Abordagem com Técnica Estatística

Técnicas estatísticas também podem ser aplicadas para selecionar boas otimizações. Uma abordagem que utiliza técnica estatística para selecionar um conjunto de otimizações com objetivo de aumentar a velocidade de execução foi apresentada por Haneda et al. (2005) e utiliza inferência estatística não paramétrica. Ela se baseia no teste de hipótese de Mann-Whitney para prever o efeito de determinada otimização durante a compilação. Nesse trabalho, são definidas as seguintes hipóteses para cada otimização:

(H0) Hipótese nula A otimização `A` não influencia a compilação do programa `B`.

(H1) Hipótese experimental A otimização `A` influencia a compilação do programa `B`.

A ideia principal desse trabalho é decidir se cada otimização A_i deve ser ativada ou não. Para isso são executados vários conjuntos de otimizações, organizados de uma maneira

pré-estabelecida e coletados seus tempos de execução. Em seguida, cada conjunto recebe um *rank*, de modo que o menor tempo recebe *rank* 1, o segundo menor tempo recebe o *rank* 2 e assim por diante.

A partir de então, o objetivo é verificar o efeito de uma otimização O_i mediante um teste de hipótese. Para alcançar esse objetivo são construídos dois grupos: um grupo experimental e um grupo de controle. O grupo experimental conterá todos conjuntos em que O_i mediante a e o grupo de controle conterá todos conjuntos em que O_i está inativa. Com esses dois grupos definidos, é aplicado o teste de Mann-Whitney (Hollander e Wolfe, 1999).

Para aplicação desse teste são somados os *ranks* de ambos grupos, obtendo um total T_1 do grupo experimental e um total T_2 do grupo de controle. Inicialmente supõe-se que a otimização em questão O_i não afeta significativamente o código, ou seja, supõe-se que a hipótese nula é verdadeira. T_1 terá no mínimo $1+2+...+N$ e no máximo $(N+1)+...+(2N)$. Foi mostrado por Hollander e Wolfe (1999) que, se a hipótese nula é verdadeira, T_1 tem uma distribuição normal com média:

$$\mu = \frac{N(2N+1)}{2}$$

e desvio padrão:

$$\sigma = \sqrt{\frac{N^2(2N+1)}{12}}$$

Para o teste de Mann-Whitney, T_1 deve estar normalizado como a seguir:

$$z = \frac{T_1-\mu}{\sigma}$$

Isto é, z mede quão longe T_1 está da média em unidades de desvio padrão. Assim, z também segue distribuição normal, que é dada por:

$$Y(z) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}z^2}$$

É também necessário considerar que a distribuição normal tem a característica definida pela expressão a seguir:

$$\int_{-\infty}^{\infty} Y(z)dz = 1$$

Com essas informações e baseando-se no teste de Mann-Whitney, deve ser considerada a questão da diferença entre T_1 e μ. Se o valor de T_1 é significativamente diferente de μ, o teste conclui que a hipótese nula é falsa. Ainda, para decidir se T_1 é significativamente diferente de μ, uma função é dada:

$$P(t) = (1 - 2 \times \int_0^t Y(z)dz) \times 100\%$$

Um critério padrão apresentado por Hollander e Wolfe (1999) para decidir a "diferença significativa", é quando $P(t)$ é menor que 5%. Isso significa que a probabilidade de rejeitar a hipótese nula quando ela é verdadeira é menor que 5%.

Todo o processo apresentado é repetido para cada otimização para determinar se ela afeta significativamente ou não o código. Se houve uma degradação de desempenho significativa, o algoritmo proposto retira a otimização do conjunto.

Para validação experimental do trabalho de Haneda et al. (2005), foram utilizados alguns dos *benchmarks* do SPEC2000, sendo sete do SPEC2000int e dois do SPEC2000fp, um total de dez programas. Os resultados foram comparados com os diferentes níveis de otimização do compilador GCC versão 3.3.1 (`-O0`, `-O1`, `-O2` e `-O3`) e também com diferentes tamanhos dos conjuntos experimental e de controle. O tamanho do conjunto de controle sempre é igual ao tamanho do conjunto experimental. Considerando n o tamanho do conjunto experimental e $N = 2n$ o tamanho dos dois conjuntos, o experimento foi executado para $N = 24, 48, 100$. O número de iterações necessárias variou de acordo com um parâmetro do algoritmo, mas foi no máximo 11 iterações.

Os resultados obtidos são apresentados na Figura 3.1, onde todos *speedups* são apresentados em relação à `-O0`.

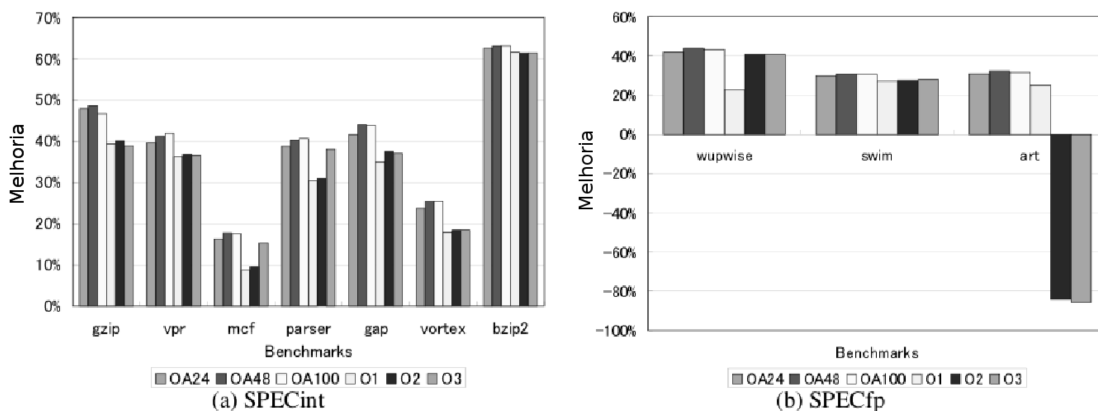


Figura 3.1: *Speedups*, em relação à O0, da aplicação da técnica estatística de Haneda et al. (2005)

3.3 Abordagens com Eliminação Iterativa

Os algoritmos propostos por Pan e Eigenmann (2006) procuram eliminar todas otimizações que afetam negativamente o desempenho do código. Assim, eles recebem como entrada um conjunto padrão a partir do qual serão retiradas as otimizações possivelmente nocivas (OPNs).

O primeiro algoritmo, o *Batch Elimination* (BE), utiliza a abordagem mais agressiva. Ele elimina toda otimização que isoladamente, não importando o quanto, prejudique o código. Essas otimizações são determinadas da seguinte forma: inicialmente ele avalia o conjunto padrão e em seguida desabilita somente a primeira otimização dela e avalia esse conjunto novamente. Depois, ele desabilita apenas a segunda otimização e faz uma nova análise. Esse processo é repetido para cada otimização e, baseado na avaliação, se o desempenho do conjunto sem a otimização for melhor do que com ela, ela será retirada do conjunto padrão.

Com essa característica, o BE apresenta-se como um algoritmo guloso, pois retira do conjunto padrão qualquer otimização que prejudique o desempenho. Tal estratégia pode não ser uma boa solução pois a análise de desempenho de cada transformação isoladamente não indica necessariamente que ela é prejudicial, isto pelo fato de uma otimização poder influenciar em outra. Portanto, os autores propuseram um segundo algoritmo, o *Iterative Elimination* (IE), que faz uma análise mais criteriosa antes de eliminar as otimizações.

O IE avalia o desempenho do conjunto padrão sem cada uma das otimizações, isto é, idêntico ao BE. Porém, após essa análise, apenas a otimização que mais prejudicou o código é retirada do conjunto padrão e todo o processo é repetido até que não haja mais otimizações que prejudiquem o código, considerando que, na próxima avaliação o novo conjunto padrão será o conjunto sem a pior otimização da etapa anterior.

A estratégia do IE, apesar de ser mais criteriosa que a do BE, tem um custo mais elevado. De fato o IE possui complexidade quadrática pois a cada n avaliações será retirada apenas uma otimização nociva, ao contrário do BE que retira todas.

Considerando que retirar as otimizações com poucos critérios (BE) pode não chegar a boas soluções e que também a complexidade do algoritmo cresce para uma busca mais criteriosa (IE), os autores propuseram uma terceira abordagem, que combina características de ambos. Por esse motivo, essa abordagem é conhecida como *Combined Elimination* (CE).

O CE, assim como o BE e o IE, avalia o desempenho do conjunto padrão sem cada uma das otimizações. Porém, a cada análise de todas otimizações, o algoritmo cria uma lista contendo todas as otimizações nocivas. Após a geração da lista, a mais nociva é

retirada dela, gerando um novo conjunto que é analisado novamente, ou seja, seu tempo de execução é calculado. Em seguida, todas as demais otimizações da lista são avaliadas, agora em relação ao conjunto do passo anterior e não da etapa inicial, e todas as que influenciam negativamente o código são retiradas. O algoritmo termina quando não há mais otimizações consideradas nocivas. O comportamento desse algoritmo faz com que seu limitante superior seja igual ao do IE, $O(n^2)$.

Para verificar a eficiência dos algoritmos, os autores fizeram uma avaliação experimental em uma máquina SPARC II, utilizando 38 otimizações do GCC (Stallman e DeveloperCommunity, 2009). O conjunto padrão utilizado foi a opção `-O3` do GCC e os *benchmarks* foram selecionados do SPEC CPU2000, sendo 11 do SPECfp e 12 do SPECint. Por fim, o ganho de desempenho médio em relação a `-O3` alcançado com essa configuração pelo CE foi de $4,1\%$ para o SPECfp e de $3,9\%$ para o SPECint. O BE ocasionou em média perda de desempenho para ambos conjuntos *int* e *fp*. O IE alcançou $4,1\%$ para o SPECfp e $3,6\%$ para o SPECint.

3.4 Abordagens com Algoritmos Genéticos

Algoritmos genéticos (AGs) foram utilizados por Almagor et al. (2004) para selecionar otimizações visando pelo menos três objetivos: reduzir número de instruções executadas, tamanho do código e, até mesmo, consumo de energia. Nesse trabalho, os autores executaram duas etapas: a enumeração e a exploração.

A enumeração trata-se de analisar alguns subconjuntos de otimizações para um programa com sua respectiva entrada. O objetivo dessa etapa é construir uma base de informações para auxiliar o entendimento das propriedades do espaço de busca, o que dá subsídios para a escolha e configuração do algoritmo de busca a ser utilizado.

A exploração trata-se da busca propriamente dita. Ela utiliza um algoritmo de busca específico para encontrar bons conjuntos para uma variedade de programas de acordo com o objetivo especificado, podendo então confirmar as intuições feitas na etapa de enumeração. É importante mencionar que a etapa de enumeração é morosa e normalmente não é utilizada em situações cotidianas.

O algoritmo genético (AG) proposto para a etapa de exploração foi uma melhoria de um algoritmo proposto em um trabalho anterior, onde os autores apresentaram um AG para selecionar otimizações visando à redução do tamanho do código (Cooper et al., 1999). No primeiro trabalho, o algoritmo utilizou uma população de 20 indivíduos de tamanho 10 ordenados pelo valor de *fitness*. A cada geração, o algoritmo removia o pior indivíduo e mais três escolhidos aleatoriamente nos 50% piores da população. Os outros

50% da população eram utilizados para cruzamento *single-point* (Beasley et al., 1993) e os indivíduos resultantes desse cruzamento eram utilizados para substituir os removidos previamente. Todos indivíduos estavam sujeitos à mutação, com exceção do que possuía o maior *fitness*.

No trabalho mais recente, após várias experimentações com diferentes variações do AG, chegaram a uma configuração com uma população entre 50 e 100 indivíduos. Os cruzamentos também seguiam a técnica *single-point*, mas eram feitos de forma probabilística com pesos nos indivíduos. Cada indivíduo tinha o seu peso calculado proporcional ao seu valor de *fitness*. A cada geração os 10% melhores indivíduos permaneciam sem alteração e os restantes eram criados repetindo o processo de cruzar um par de indivíduos, seguido de um processo de mutação com baixa probabilidade. Além disso, se um novo indivíduo já tivesse aparecido em alguma geração anterior, ele sofria mutação até ser um indivíduo inédito.

A análise experimental realizada para verificar a eficiência do AG utilizou dez *benchmarks* de diferentes conjuntos, sendo quatro do *MediaBench* (Lee et al., 1997), dois do *SPEC95*, um do *Eratosthenes* (Collins, 1998) e três de métodos numéricos apresentados em (Forsythe et al., 1977). Para cada um dos dez programas a busca foi executada e foi tomado o melhor valor de três execuções.

O experimento foi executado com as configurações 50×50, 50×100 e 100×100, onde 50×100 significa população 50 e número de gerações 100. O conjunto padrão escolhido foi uma definido pelos próprios autores e o ganho de desempenho médio para cada uma das três configurações foi de $73,47\%$, $73,1\%$ e $72,62\%$, respectivamente. É interessante notar que o melhor resultado foi para o menor número de indivíduos e também o menor número de gerações, que correspondem a 4550 avaliações de código (iterações).

Solucionando de maneira indireta o PSO, Leather et al. (2009) utilizou programação genética para propor um mecanismo capaz de prever as características de um código e então utilizá-las em qualquer abordagem de aprendizagem de máquina.

Basicamente, a arquitetura do sistema proposto por Leather et al. (2009) inclui três componentes: o gerador de dados, o componente de programação genética (CPG) e a ferramenta de aprendizagem de máquina (AM). O gerador de dados extrai informações estáticas do código intermediário e as repassa para o CPG no formato de uma gramática. Esse componente então envia as informações das características à ferramenta de AM e recebe dela a qualidade das informações fornecidas. Assim, é construída uma população de gramáticas à qual são aplicados operadores de mutação e cruzamento.

É importante mencionar que as gramáticas desse trabalho armazenam as informações heurísticas que serão passadas ao algoritmo de AM. Assim, quanto melhor a qualidade da informação heurística, melhores oportunidades encontrará o algoritmo de AM.

Para validar a estratégia proposta, os autores aplicaram-na na otimização conhecida como *loop unroll*. Mais especificamente, o objetivo foi utilizar um algoritmo de AM em associação com o AG proposto para melhoria automática da heurística e verificar se o algoritmo de AM conseguia melhorar suas decisões quanto a que nível aplicar no *loop unroll*.

A experimentação foi realizada com o GCC 4.3.1 e em uma máquina Pentium 6, utilizando 57 *benchmarks* dos conjuntos *MediaBench*, *MiBench* e UTDSP [1]. Enquanto a heurística estática presente no código do GCC alcançou em média 3% do desempenho conhecido, os algoritmos recentes de AM alcançaram em média 59% e, quando associados a essa estratégia de geração de heurística com AGs, o ganho médio foi de 76%.

3.5 Abordagens com Aprendizagem de Máquina

Algumas abordagens aplicadas ao PSO envolvem Aprendizagem de Máquina (AM), as quais consistem basicamente em construir uma base de conhecimento e uma estratégia para fazer predições baseando-se em tal conhecimento. Vários trabalhos utilizam esta estratégia (Cavazos et al., 2007; Cavazos e O'Boyle, 2006; Park et al., 2011; Stephenson et al., 2003).

Um trabalho com o uso de AM é apresentado por Cavazos e O'Boyle (2006). Os autores apresentaram uma abordagem com AM para prever bons conjuntos de otimizações para o compilador *Just-in-time* (JIT) do Java Jikes RVM. Essa abordagem seguiu duas etapas: treino e desenvolvimento. A primeira etapa é feita uma única vez e compõe as informações base para a aprendizagem. A etapa de desenvolvimento trata do uso da heurística, que foi construída na etapa de treino, no compilador JIT.

A etapa de treino foi feita para os níveis de otimização O0, O1 e O2 do JIT/Jikes. Para os níveis O0 e O1 haviam apenas 4 e 9 otimizações, respectivamente e, portanto, foi feito um treino exaustivo. Já em O2 havia 20 otimizações, o que inviabilizou um treino exaustivo. Para este caso, foram executados 1000 conjuntos de otimizações gerados aleatoriamente. Como o objetivo do trabalho era o de aumentar a velocidade dos métodos, para cada execução foi coletado seu tempo. Esse treino foi realizado para vários métodos dos quais também foram extraídas informações para caracterização.

[1] `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`

A caracterização dos métodos foi baseada em informações como número de *bytecodes*, se o método é *final*, se o método é *private*, se o método instancia objetos, dentre outras informações.

Com as informações de características dos métodos bem como o tempo de execução deles para determinados conjuntos de otimizações, foi possível criar um modelo de predição com o uso de regressão logística. Esse modelo era então instalado no JIT da Jikes para prever um bom conjunto de otimizações dadas as características de um método.

Para validação da abordagem apresentada por Cavazos e O'Boyle (2006) foi utilizada a máquina virtual Jikes RVM, com os benchmarks DaCapo, SPECjbb 2000 e SPECjvm 98 (com entrada "large"). Foram realizados dois experimentos: um *leave-one-out cross validation* entre os programas do SPECjvm 98 e também um experimento Treino e Teste, onde o conjunto de treino foi o SPECjvm 98 e o de teste foi o DaCapo.

Os resultados do *leave-one-out* apresentaram um ganho de 1% em termos de tempo total. Já a avaliação Treino e Teste apresentou em média 4% de redução do tempo total.

Outro trabalho que utiliza AM é o de Cavazos et al. (2007), que apresentou um modelo de predição para determinar qual a probabilidade de uma otimização ser utilizada ou não. O objetivo foi utilizar *performance counters* normalizados para caracterizar o comportamento dinâmico do programa e a partir de tais identificadores utilizar conhecimento prévio para decidir qual a probabilidade de uma otimização ser ativada.

Inicialmente, foram executados 500 conjuntos de otimização para cada programa de um conjunto de *benchmarks* de treino e todos conjuntos que resultavam em um tempo de execução menor que do conjunto padrão eram inseridos em uma base de conhecimento. Essas informações eram então utilizadas em um processo de treino, de modo a construir uma função de predição cuja entrada era o conjunto de *performance counters* do programa e a saída era um vetor de probabilidades. O método utilizado para construção da função de predição foi a regressão logística.

Para avaliação experimental da proposta de Cavazos et al. (2007), foi utilizado o compilador *PathScale Ekopath* com 121 otimizações/opções de configuração. O conjunto padrão utilizado para comparação foi a opção `-Ofast` do *Ekopath*. Os *benchmarks* utilizados na experimentação foram o SPEC 95 FP, SPEC 2000 FP e INT (utilizado na etapa de treino), o *Polyhedron* 2005 e o *MiBench*.

Os resultados foram comparados com o trabalho de Pan e Eigenmann (2006), apresentando vantagens. Por fim, o *speedup* médio alcançado tanto para os programas do SPEC quanto para os outros foi de 1,17. Para alcançar tal *speedup*, foram necessárias 25 avaliações.

O trabalho de Park et al. (2011) apresentou três modelos de predição, intitulados *Sequence Predictor*, *Speedup Predictor* e *Tournament Predictor*.

Os três modelos propostos utilizam *performance counters* (PCs) para caracterizar o comportamento do programa e, de forma geral, cada modelo é uma função capaz de receber como entrada as características do programa e fornecer um ou mais conjuntos de otimizações a serem avaliados.

No primeiro modelo é analisada a viabilidade de cada otimização individualmente. Na verdade há um modelo para cada otimização e cada um deles recebe o conjunto de PCs e analisa a probabilidade da respectiva otimização ser benéfica. Vários conjuntos podem ser gerados a partir da saída desse modelo por uma seleção baseada na distribuição de probabilidade resultante das N otimizações. Para treino desse modelo, foi selecionada apenas uma otimização por *benchmark*, aquela que isoladamente fornece o maior *speedup*. Assim, os dados de treino consistiram das características de programas diversos associados às suas respectivas melhores otimizações.

O segundo modelo é treinado para prever o *speedup* de um conjunto de otimizações em relação ao conjunto padrão para um dado programa. A entrada é o conjunto de PCs do programa e um conjunto T e a saída é o *speedup* previsto. Com esse modelo é possível, por exemplo, prever o *speedup* de várias possibilidades, ou seja, vários conjuntos de otimizações sem executar o programa, executando somente uma fração reduzida desses conjuntos, geralmente os melhores previstos, para validação efetiva. Os dados de treino desse modelo consistiram de 500 conjuntos gerados aleatoriamente aplicados a cada *benchmark* associados aos respectivos *speedups* resultantes.

O terceiro modelo tem por objetivo prever qual o melhor de dois conjuntos para um determinado programa. A entrada desse modelo consiste do conjunto de PCs do programa e de dois conjuntos de otimizações. A saída é a diferença de *speedup* prevista entre os dois conjuntos. Além disso, esse modelo fornece não apenas o melhor conjunto de dois, mas também o quanto um é melhor que o outro. Esse modelo pode gerar conjuntos para um novo programa por meio da classificação de diferentes conjuntos gerados por operações par a par, no formato de um torneio, como o nome sugere.

Além de analisar esses três modelos de predição, o trabalho também considerou dois algoritmos de AM. O primeiro foi o Support Vector Machines (SVMs) (Chang e Lin, 2011), um algoritmo de aprendizagem supervisionada e o segundo foi o clássico algoritmo de regressão linear (Draper e Smith, 1998).

Para análise experimental dos modelos propostos e dos algoritmos de AM escolhidos, foi utilizado o compilador Open64 4.2.1 [2], o *HPCToolKit* (Adhianto et al., 2010) para

[2]`http://www.open64.net`

coleta de PCs e o Weka (Markov e Russell, 2006) como ferramenta de AM, que suporta os dois algoritmos selecionados. O conjunto padrão foi a opção `-Ofast` do Open64 e cerca de 64 *benchmarks* foram selecionados dos conjuntos UTDSP, NPB (Bailey et al., 1991), *Linpack* (Dongarra et al., 2003) e *PolyBench*, incluindo funções e aplicações *kernel*, sendo executado um experimento *leave-one-out* entre os 64 *benchmarks*. O número de conjuntos experimentados foi 10 para todos os modelos. Além disso, foi feito um experimento considerando os 64 programas como o conjunto de treino e o *MiBench* como conjunto de teste.

O *speedup* médio alcançado na experimentação *leave-one-out* com a regressão linear foi de $1,61$ para o *Sequence Predictor*, $1,69$ para o *Speedup Predictor* e de $1,75$ para o *Tournament Predictor*. Com o uso do algoritmo SVM, os *speedups* foram de $1,63$, $1,68$ e $1,75$, respectivamente. Na experimentação feita exclusivamente com o *MiBench*, o ganho de desempenho médio foi de 10%.

Mais recentemente, Kulkarni e Cavazos (2012) apresentaram uma estratégia com AM para selecionar automaticamente uma melhor ordem de aplicação das otimizações. A abordagem foi desenvolvida no compilador JIT Jikes RVM Java [3], utilizando Redes Neurais Artificiais (RNAs) (Russell et al., 1996). As RNAs utilizadas no trabalho foram induzidas automaticamente com o uso do *Neuro-Evolution for Augmenting Topologies* (NEAT) (Stanley e Miikkulainen, 2002).

Assim como outros trabalhos de seleção de otimizações com AM, foram utilizadas as características do código para prever o conjunto de otimizações, mais especificamente a ordem de aplicação destas. As características do código eram estáticas, no sentido de serem retiradas de um código já compilado. No entanto, o processo de coletar as características era feito para cada método individualmente e a cada vez que ele era executado.

Dois modelos de predição foram potenciais candidatos: um que previa uma sequência de otimizações completa dadas as características do código e outro que previa a melhor otimização a ser aplicada no estado corrente do método. Os autores escolheram o segundo modelo por acreditarem ser um problema de aprendizagem mais fácil de solucionar.

Inicialmente, uma RNA gerada pelo NEAT é carregada no *driver* de otimização da Jikes RVM. Em seguida, as seguintes etapas são executadas para cada método selecionado para compilação:

1. Gerar um vetor de características do estado atual do método;

2. Utilizar a RNA para prever a melhor otimização a ser aplicada;

[3] `http://jikesrvm.org`

3. Saltar para a primeira etapa destas três até que a previsão da RNA seja não adicionar mais otimizações.

Após a repetição dessas etapas, restará obter o *speedup* do código gerado e enviá-lo como um *feedback* ao NEAT.

Todo o processo descrito acima é utilizado para busca da melhor RNA. Esse processo é moroso mas deve ser feito, teoricamente, uma única vez. Assim que essa RNA é encontrada, ela é introduzida no compilador Jikes RVM e utilizada em tempo de execução como uma heurística de otimização.

O NEAT utiliza um algoritmo genético para gerar as RNAs. Inicialmente, são geradas 60 RNAs e avaliadas nos *benchmarks* do conjunto de treino do experimento e apenas as dez melhores são passadas para a próxima geração. As próximas gerações são construídas com mutação e cruzamento sobre essas melhores RNAs. As mutações podem envolver a adição de neurônio, adição de aresta ou remoção de aresta.

Para avaliação experimental da heurística construída, foram utilizados os conjuntos de *benchmarks* SPECjvm98/SPECjvm2008 e DaCapo (Blackburn et al., 2006). Todos *benchmarks* do SPECjvm98 e um subconjunto do SPECjvm2008 e do DaCapo que compilaram corretamente com o Jikes RVM.

O objetivo da avaliação foi medir o ganho de desempenho em termos de tempo de execução. O ganho médio em relação à opção `-O3` do compilador foi de 10% para o SPECjvm98, 7% para o SPECjvm2008 e de 7,3% para o DaCapo.

3.6 Abordagens Híbridas

Lau et al. (2006) apresentou um trabalho de avaliação de desempenho *online* da *Java Virtual Machine* (JVM) (Lindholm e Yellin, 1999), que combina técnicas estatísticas e aleatoriedade. O *framework* apresentado é chamado Auditor de Desempenho e possibilita uma análise de desempenho *online*, isto é, efetua as análises enquanto o programa está executando. Essa análise é feita por regiões de código.

Em compiladores *Just-in-time* (JIT) (Perez et al., 2012), são identificadas regiões de código frequentemente utilizadas, chamadas *hot regions* (Kumar et al., 2002), as quais são compiladas. Por meio da avaliação de desempenho *online* do Auditor de Desempenho, o objetivo foi prever boas versões de código compilado para uma dada região. Para isso, são geradas N versões de código para uma mesma região. Segundo os autores, “a ideia chave é selecionar aleatoriamente uma dessas implementações (versões) sempre que se entra em uma região de código e gravar quanto tempo foi gasto na execução”.

Após a coleta de tempo de várias versões de código, escolhidas aleatoriamente pelo Auditor de Desempenho, um mecanismo estatístico determina qual conjunto de tempos de execução é o melhor. Essa escolha é desafiadora porque a execução de uma mesma região ocorre com o programa em diferentes estados (parâmetros e/ou valores de memória). Por exemplo, uma versão A de uma região de código pode executar em menos tempo que uma versão B da mesma região, não porque A tem uma implementação melhor, mas porque A executa a partir de um estado que o faz realizar menos trabalho. Por isso, os autores propõem um mecanismo estatístico para determinar a melhor versão.

O trabalho de Purini e Jain (2013) utilizou algoritmos genéticos, busca aleatória, eliminação iterativa e aprendizagem de máquina combinados para buscar soluções para o PSO.

O objetivo desse trabalho foi encontrar conjuntos de otimizações que cobriam o maior número de classes de programas. Para este fim foi criado um espaço de busca reduzido que deveria conter ao menos um bom conjunto de otimizações para um determinado programa de entrada.

Para construção desse espaço de busca foram utilizadas seis técnicas diferentes. Três delas são as seguintes:

Algoritmo genético com um seletor por *rank* que produz 100 gerações e utiliza como critério de parada a não convergência durante três gerações consecutivas ou se o *fitness* dos indivíduos de determinada geração apresentar um desvio padrão menor que 0,01;

Algoritmo genético com um seletor por torneio cuja estratégia é idêntica à anterior, exceto pelo fato do seletor ser diferente;

Busca aleatória uniforme que retorna o melhor conjunto a partir de 500 conjuntos gerados aleatoriamente.

As outras três técnicas são as três citadas acima com a diferença de que cada conjunto gerado foi ajustado de forma a obedecer as recomendações de ordem fornecidas no manual do compilador em questão (Lattner e Adve, 2004).

Foram utilizadas 61 aplicações *Microkernel* e o espaço de busca foi construído com cada uma dessas aplicações com cada uma das seis técnicas acima citadas. Após esta etapa, 366 conjuntos de otimizações foram gerados. Eliminando as redundâncias, chegaram a um espaço final de 290 conjuntos. Além disso, também foi aplicada uma técnica para eliminar as otimizações que prejudicavam o código, sendo que o espaço de busca foi reduzido a, pelo menos, 80% após essa etapa.

Foram propostos então três algoritmos para seleção de conjuntos de otimização, chamados pelos autores de *Best-10*, *Best-12* e *Cluster-10*.

O *Best-10* extrai do espaço de busca os dez melhores conjuntos, da seguinte forma: considerando P inicialmente o conjunto de programas, ele procura qual o conjunto de otimizações que dá *speedup* para o maior número de programas de P e o adiciona na lista dos dez melhores. Em seguida ele retira de P todos os programas para os quais o conjunto de otimizações selecionado dá *speedup*. Considerando o novo conjunto P, o processo é repetido até selecionar os dez melhores conjuntos.

O *Best-12* executa o processo do *Best-10*, toma de seu resultado os seis melhores conjuntos e os retira do espaço de busca. O processo do *Best-10* é repetido com esse novo espaço de busca e os seis melhores conjuntos são selecionados dele. Assim, a saída do *Best-12* é o conjunto com os seis melhores conjuntos da primeira etapa juntamente com os seis melhores da segunda etapa.

O *Cluster-10* utiliza um algoritmo de clusterização para dividir os conjuntos de otimizações do espaço de busca em dez grupos, de modo que cada grupo contenha conjuntos similares entre si. A partir de então, o processo do *Cluster-10* toma inicialmente um *cluster* aleatoriamente e verifica qual conjunto nele cobre o maior número de programas no espaço de busca. Similar ao *Best-10*, ele retira do conjunto P os programas cobertos e repete o processo tomando outro *cluster*. Esse processo é feito até completar o total de dez conjuntos de otimização.

A experimentação que avaliou a abordagem de (Purini e Jain, 2013) utilizou a infraestrutura de compilação LLVM e os *benchmarks MiBench* e *PolyBench*. O conjunto padrão utilizado foi a opção `-O2` da LLVM. As abordagens avaliadas foram: algoritmos genéticos, todas 290, *Best-10*, *Best-12* e *Cluster-10*. No conjunto *MiBench* apresentaram ganho de desempenho de $11,2\%$, $16,7\%$, $8,73\%$, $9,7\%$ e $9,8\%$, respectivamente. No conjunto *PolyBench* as mesmas abordagens apresentaram ganho de desempenho de $10,87\%$, $13,9\%$ $11,1\%$, $11,37\%$ e $11,17\%$, respectivamente.

3.7 Considerações Gerais

Este capítulo apresentou diversas propostas encontradas na literatura para a solução do PSO. Inicialmente foram apresentados trabalhos com busca exaustiva com o objetivo de mostrar que ela pode ser utilizada em alguns contextos, ainda que bem específicos. Nenhuma das abordagens propostas no presente trabalho utilizou técnica de busca exaustiva.

Uma estratégia mais elaborada que as exaustivas foi apresentada posteriormente e utilizou técnicas estatísticas, mais especificamente um teste de hipótese, para selecionar otimizações. Esse trabalho se assemelha a três abordagens propostas aqui pelo fato de utilizarem probabilidade e similaridade estatística. No entanto, diferente da abordagem exposta neste capítulo, as abordagens propostas não utilizam testes de hipótese.

Outras estratégias alcançaram ganho de desempenho efetuando eliminações de otimizações provavelmente nocivas (OPNs), sendo chamadas de abordagens de eliminação iterativa. Dois algoritmos propostos neste trabalho se basearam em tais estratégias no sentido de também procurar OPNs para remoção. No entanto ao invés de utilizar uma abordagem determinística para isso, os algoritmos propostos utilizam uma abordagem probabilística.

O uso de algoritmos genéticos (AGs), que são também metaheurísticas, foi tanto aplicado diretamente à solução do PSO quanto como uma ferramenta de apoio a ferramentas de aprendizagem de máquina. Já as abordagens aqui propostas não utilizaram nenhuma estratégia envolvendo AGs, porém uma delas utilizou metaheurística.

Trabalhos com aprendizagem de máquina (AM) também foram apresentados e se destacam, principalmente, por conseguirem alcançar *speedups* com um número muito baixo de avaliações de código. Dentre as estratégias propostas há uma que utiliza a técnica de AM chamada Raciocínio Baseado em Casos (RBC) e também é proposta com objetivo de selecionar boas otimizações com um número inferior de avaliações.

Por fim, foram apresentados dois trabalhos que utilizaram uma combinação de técnicas para seleção de otimizações. O primeiro combinou aleatoriedade e técnicas estatísticas ao passo que o segundo utilizou pelo menos quatro técnicas diferentes para solução. Algumas abordagens propostas no presente trabalho também utilizam múltiplas técnicas. Duas delas combinam eliminação iterativa com técnicas estatísticas sendo que uma dessas duas utiliza também aleatoriedade. Uma outra abordagem entre as propostas utiliza AM e também estratégia aleatória para construção de um espaço de busca inicial. A Tabela 3.1 apresenta uma breve comparação entre os trabalhos relacionados e as propostas do presente trabalho.

No capítulo seguinte serão apresentadas em detalhes as estratégias propostas e implementadas no presente trabalho que visam à seleção de otimizações com objetivo primordial de alcançar *speedups*.

Trabalho	Estratégia	Semelhanças	Diferenças
Foleiss et al. (2011a)	busca exaustiva	resolução do PSO	aplicação em tamanho de código
Kulkarni et al. (2006)	busca exaustiva	resolução do problema de ordem	somente enumera o espaço de busca
Haneda et al. (2005)	técnica estatística	técnica estatística é utilizada no *CBR-Selector*	no *CBR-Selector* não é utilizado teste de hipótese
Pan e Eigenmann (2006)	eliminação iterativa	eliminação iterativa é utilizada no PBE	o PBE utiliza abordagem probabilística
Almagor et al. (2004)	algoritmo genético	utilização de metaheurística	a metaheurística proposta neste trabalho foi a VNS
Leather et al. (2009)	algoritmo genético	utilização e metaheurística	a metaheurística proposta neste trabalho foi a VNS
Cavazos e O'Boyle (2006)	aprendizagem de máquina	uso de aprendizagem	a técnica utilizada foi Regressão Logística e no presente trabalho Raciocínio Baseado em Casos (RBC)
Cavazos et al. (2007)	aprendizagem de máquina	uso de aprendizagem	a técnica utilizada foi Regressão Logística e no presente trabalho RBC
Park et al. (2011)	aprendizagem de máquina	uso de aprendizagem	a técnica utilizada foi Regressão Logística e SVM; e no presente trabalho RBC
Kulkarni e Cavazos (2012)	aprendizagem de máquina	uso de aprendizagem	foi aplicado no contexto da ordem de aplicação de otimizações e em compilação JIT
Lau et al. (2006)	híbrida	utiliza em parte da estratégia busca aleatória	aplicado no contexto de compilação JIT
Purini e Jain (2013)	híbrida	geração de um espaço de busca prévio	criação de novos conjuntos-padrão de otimizações

Tabela 3.1: Comparação entre as estratégias utilizadas nos trabalhos relacionados com as estratégias propostas

4

Abordagens Propostas para Mitigar o Problema da Seleção de Otimizações

Este capítulo apresenta quatro abordagens propostas para o PSO, a saber:

- Uma abordagem que utiliza metaheurística clássica;
- Duas abordagens baseadas no algoritmo *Batch Elimination*, apresentado no Capítulo 3;
- Uma abordagem que utiliza AM, mais especificamente a técnica de raciocínio baseado em casos.

As seções seguintes apresentam a descrição detalhada de cada uma das abordagens propostas. Inicialmente é apresentada a estratégia que utiliza a metaheurística clássica VNS, seguida dos algoritmos baseados na eliminação iterativa e, por fim, a abordagem com Raciocínio Baseado em Casos (RBC).

4.1 Variable Neighborhood Search

Variable Neighborhood Search (VNS) é uma metaheurística proposta em 1995 por Mladenović (1995) e amplamente utilizada para resolução de problemas combinatórios e de otimização global. Basicamente, esta metaheurística efetua mudanças sistemáticas na vizinhança de uma solução com uma busca local para encontrar novas soluções para o problema (Gendreau e Potvin, 2010). Assim, um algoritmo que utiliza a metaheurística VNS segue o seguinte comportamento, em ordem:

Inicialização Definir uma solução inicial x, uma condição de parada e um conjunto de operadores que serão utilizados na busca. Operadores são métodos que modificam uma solução para buscar soluções vizinhas a ela, seguindo critérios predeterminados. A solução inicial é o ponto de partida para a busca de novas soluções;

Repetição Repetir os seguintes passos, até que acabem os operadores definidos na inicialização:

Perturbação gerar uma solução x' vizinha de x de forma aleatória;

Busca local aplicar uma busca local com o operador atual, utilizando x' como solução inicial e encontrar um ótimo local x'';

Trocar ou não Se o ótimo local x'' for melhor que a solução x então fazer $x \leftarrow x''$ e prosseguir na iteração com o mesmo operador; senão, selecione o próximo operador;

Alguns problemas combinatórios conhecidos já tiveram a aplicação da VNS, como: problema do escalonador de processos flexível (Zhang, 2012); problema de fluxo de montagem com máquinas paralelas (Javadian et al., 2009); problema do caixeiro viajante (Lei-fu e Wei, 2010); e problema de escalonamento de horários (Saviniec e Constantino, 2013).

As características do PSO permitem que ele também seja abordado com a metaheurística VNS. Portanto, essa é uma das propostas do presente trabalho e é descrita nas subseções seguintes.

4.1.1 Operadores de Vizinhança

Os operadores de vizinhança norteiam a busca local. Para o PSO, é interessante explorar não só quais otimizações estarão no conjunto mas também em que ordem elas estarão. Para explorar ainda mais o espaço de busca, é possível permitir a repetição de aplicação de otimizações. Para atingir todos estes objetivos, foram definidos os seguintes operadores de vizinhança:

Troca seleciona duas otimizações aleatoriamente e troca uma pela outra;

Remoção escolhe uma otimização de forma aleatória ou arbitrária e a remove;

Adição escolhe aleatoriamente uma otimização de uma tabela de otimizações T e a adiciona na sequência, em uma posição aleatória.

O operador de troca explora a ordem de aplicação das otimizações, enquanto os operadores de adição e remoção exploram quais otimizações estarão na solução. Nenhuma restrição é feita na adição quanto a qual otimização está sendo adicionada. Assim, é possível que uma mesma otimização seja adicionada mais de uma vez.

4.1.2 Rotina de Perturbação

A rotina de perturbação tem por objetivo explorar outra região de soluções, ampliando desta forma as chances de encontrar um ótimo global.

Para alcançar este objetivo, a rotina de perturbação implementada escolhe um operador de forma aleatória e aplica-o uma única vez na solução que deve ser perturbada, conforme mostra o Algoritmo 1.

Algoritmo 1: Rotina de perturbação da VNS

Dados: Solução a ser perturbada (S)
Resultado: Solução perturbada (S)
procedimento PERTURBACAO(S)
  $lista_operadores \leftarrow [TROCA, REM, INS]$;
  $OP \leftarrow selecionar_aleatorio(lista_operadores)$;
  Aplique o operador OP em S;
  retorne S
fim procedimento

4.1.3 Busca Local

A busca local tem por objetivo procurar a melhor solução em uma certa vizinhança. Isso pode ser feito explorando toda a vizinhança, ou seja, explorando todas as soluções que podem ser geradas a partir de uma determinada solução com um determinado operador. No entanto, explorar todas essas possibilidades possui um alto custo pois envolve, em geral, a execução do programa a cada avaliação. Portanto, uma estratégia menos dispendiosa é explorar um número limitado de soluções em cada vizinhança. Nesta implementação isto é feito explorando l soluções da vizinhança, onde l é o tamanho da solução sobre a qual deve ser feita a busca local.

Seguindo tal estratégia, foram desenvolvidas três buscas locais, que são apresentadas nos Algoritmos 2, 3 e 4, com os operadores troca, remoção e adição, respectivamente.

A busca local de troca efetua uma troca aleatória na sequência e a avalia. Se a troca for benéfica, então essa solução é utilizada na próxima troca aleatória. Esse processo

Algoritmo 2: Busca local com o operador "troca"

Dados: Solução para a qual deve ser analisada a vizinhança (S); Tempo da solução S (T);
Resultado: Melhor solução encontrada na vizinhança (S) e seu tempo de execução (T)

```
procedimento BUSCA_LOCAL_TROCA(S, T)
    repita
      tempo_inicio ← T;
      l ← tamanho(S);
      para i = 1 → l faça
          Aplique o operador de troca em S, obtendo S';
          T' ← execute(S');
          se T' ≤ T então
              S ← S';
              T ← T';
          fim
      fim
    até que T ≥ tempo_inicio;
    retorne S, T
fim procedimento
```

é repetido l vezes. A busca finaliza assim que saltar para uma solução que degrade o desempenho. Assim, esse operador tem por objetivo melhorar a solução explorando a ordem de aplicação das otimizações.

A busca local de remoção retira uma otimização de S, gerando S'. O desempenho de S' é então avaliado e, se S' tem desempenho superior a S, uma próxima remoção será realizada em S'. Basicamente, a busca local de remoção remove otimizações enquanto houver melhoria em relação à solução anterior. Isso explora a eliminação de otimizações que afetam negativamente o desempenho.

A busca local de adição insere até l novas otimizações na solução inicial. A posição de inserção da otimização é arbitrária, enquanto a otimização a ser inserida é aleatória. A inserção é feita iterativamente da posição 1 até a l. Isso permite a inserção de otimizações em diferentes posições, além de permitir a repetição de aplicação de uma mesma otimização. Esses fatores ampliam o espaço de busca fornecendo novas possibilidades para encontrar boas soluções para o PSO.

Algoritmo 3: Busca local com o operador "remoção"

Dados: Solução para a qual deve ser analisada a vizinhança (S); Tempo da solução S (T);

Resultado: Melhor solução encontrada na vizinhança (S) e seu tempo de execução (T)

```
procedimento BUSCA_LOCAL_REM(S, T)
    repita
        tempo_inicio ← T;
        l ← tamanho(S);
        para i = 1 → l faça
            Aplique o operador de remoção em S, removendo a otimização da
        posição i da solução S, obtendo S';
            T' ← execute(S');
            se T' ≤ T então
                S ← S';
                T ← T';
            fim
        fim
    até que T ≥ tempo_inicio;
    retorne S, T
fim procedimento
```

Algoritmo 4: Busca local com o operador "adição"

Dados: Solução para a qual deve ser analisada a vizinhança (S); Tempo da solução S (T);

Resultado: Melhor solução encontrada na vizinhança (S) e seu tempo de execução (T)

```
procedimento BUSCA_LOCAL_INS(S, T)
    repita
        tempo_inicio ← T;
        l ← tamanho(S);
        para i = 1 → l faça
            Aplique o operador de adição em S, adicionando uma otimização
        aleatória na posição i da solução S, obtendo S';
            T' ← execute(S');
            se T' ≤ T então
                S ← S';
                T ← T';
            fim
        fim
    até que T ≥ tempo_inicio;
    retorne S, T
fim procedimento
```

4.1.4 Algoritmo VNS para o PSO

Tendo definidos a rotina de perturbação, os operadores de vizinhança e as buscas locais correspondentes, é possível construir um algoritmo com a metaheurística VNS, definindo o critério de parada e a solução inicial. O algoritmo aqui proposto utiliza número de iterações como critério de parada, isto é, todo o processo da VNS é repetido N vezes. Quanto maior for este número, é esperado que os resultados sejam melhores, porém ao custo de um maior tempo de execução do algoritmo. A solução inicial pode ser qualquer conjunto, desde um gerado aleatoriamente ou até mesmo o conjunto padrão. A VNS para o PSO pode ser vista no Algoritmo 5.

Algoritmo 5: VNS para o PSO

```
Dados: Programa a ser avaliado (P); Conjunto inicial (S); Número de repetições
       (N)
Resultado: Melhor conjunto de otimizações encontrado (melhor_conj)
  procedimento Busca_VNS(P, S, N)
     lista_operadores ← [TROCA, REM, INS];
     OP ← primeiro(lista_operadores);
     (conj, t) ← busca_local(S, OP);
     melhor_t ← t;
     melhor_conj ← conj;
     para i = 1 → N faça
      enquanto Não terminaram os operadores faça
         (conj, t) ← perturbacao(conj);
         (conj, t) ← busca_local(conj, OP);
         se t < melhor_t então
            melhor_conj ← conj;
            melhor_t ← t;
            Vá para o primeiro operador;
          fim
         senão
            conj ← melhor_conj;
            Vá para o próximo operador;
          fim
       fim
      fim
     retorne melhor_conj;
  fim procedimento
```

Por efetuar o tipo de busca mais completo entre as abordagens descritas neste trabalho, isto é, uma busca que envolve seleção de otimizações, ordem e repetições, o uso da VNS

para o PSO parece ser uma boa abordagem. No entanto, tal abordagem busca soluções sem considerar aspectos específicos do PSO como, por exemplo, o fato de que nem sempre uma otimização aumenta o desempenho.

As próximas duas seções expõem outras abordagens para o PSO, as quais consideram características específicas do problema para fazer a busca dos conjuntos de otimizações.

4.2 Probabilistic Batch Elimination

Em certo sentido, os algoritmos IE e CE apresentados na subseção 3.3 repetem o processo de "compilar o conjunto padrão sem cada uma das otimizações e avaliar o resultado de cada caso". Afinal, o conjunto padrão é alterado entre essas repetições, o que torna necessário repetir o processo para procurar as otimizações provavelmente nocivas (OPNs). O BE, também apresentado na subseção 3.3, evita esta repetição, removendo todas as OPNs em um único passo, considerando apenas cada uma de forma isolada. No entanto, é possível evitar tal repetição e ainda assim utilizar mais informações para decisão de quais otimizações remover. Essa é a proposta do *Probabilistic Batch Elimination* (PBE).

O PBE é resultante de uma modificação no BE. Enquanto o BE utiliza uma abordagem agressiva para decidir quais otimizações serão removidas, o PBE utiliza uma abordagem probabilística. Diferente do BE, no PBE cada otimização que prejudica o código tem um fator de nocividade calculado. Intuitivamente, é possível dizer que para cada programa algumas otimizações prejudicam menos e outras prejudicam mais o desempenho. Assim, o PBE utiliza tal hipótese e considera que as otimizações que prejudicam mais o código têm uma maior probabilidade de serem removidas do conjunto padrão enquanto as que prejudicam menos o código têm uma menor probabilidade. Portanto, a probabilidade de uma determinada otimização ser removida será diretamente proporcional ao seu fator de nocividade. O fator de nocividade de uma otimização será 0 se o desempenho do conjunto padrão com ela é melhor que o do conjunto padrão sem ela. Caso contrário o fator de nocividade será o tempo de execução do conjunto sem a otimização em questão.

O cálculo das probabilidades é feito da seguinte forma: a otimização mais nociva recebe probabilidade 1, ou seja, a otimização mais nociva sempre é eliminada. As outras otimizações recebem probabilidades proporcionais. Por exemplo, se a otimização mais nociva tem fator 4,9, uma segunda otimização, com fator 3,5, terá como probabilidade o valor $\frac{3,5}{4,9} = 0,95$.

Após o cálculo da probabilidade de todas as otimizações nocivas, é gerado um número aleatoriamente e todas as otimizações com probabilidade maior ou igual a tal número são removidas do conjunto padrão.

Algoritmo 6: *Probabilistic Batch Elimination*

```
Dados: Programa (P); Conjunto Padrão (B); Número de tentativas (c)
Resultado: Melhor conjunto de otimizações encontrado (melhor_conj)
  procedimento PBE(P, B, c)
      tam_B ← tamanho(B);
      melhor_conj ← B;
      melhor_tempo ← execute(P, B);
      para i = 1 → c faça
        prob ← crie uma nova lista vazia;
        conj = B;
        para i = 1 → tam_B faça
            Insira 0 ao final da lista prob;
        fim
        para i = 1 → tam_B faça
            Ligue todas as otimizações em B;
            Desligue a otimização i em B;
            tempo_i ← execute(P, B);
            se tempo_i > tempo_B então
                prob[i] ← tempo_i
            fim
        fim
        M ← maximo(prob);
        se M diferente de 0 então
            para i = 1 → tam_B faça
                prob[i] ← prob[i] / M;
            fim
        fim
        R ← gere um número aleatório entre 0 e 1;
        para i = 1 → tam_B faça
            se conj[i] ≥ R então
                Desligue a otimização i em conj;
            fim
        fim
        tempo_t ← execute(P, conj);
        se tempo_t < melhor_tempo então
            melhor_tempo ← tempo_t;
            melhor_conj ← conj;
        fim
      fim
      retorne melhor_conj;
  fim procedimento
```

Para ampliar as chances, o PBE possibilita que todo o processo descrito acima seja repetido c vezes para decidir qual o melhor conjunto de otimizações dentre as tentativas. O Algoritmo 6 descreve o PBE.

Com essas características, o PBE é um algoritmo com complexidade $\Theta(cn)$, ou seja, dependendo do número de tentativas executará mais rápido do que os algoritmos IE e CE e, por fornecer uma estratégia probabilística, amplia as oportunidades de buscar conjuntos de otimizações diferentes.

4.3 Improved Probabilistic Batch Elimination

Apesar de o PBE funcionar de forma que tempo de resposta *versus* qualidade do resultado possam ser definidos pelo usuário, ele apresenta alguns pontos que podem ser explorados de forma mais eficiente.

Um problema do PBE é a repetição do processo de cálculo do fator de nocividade. Isso faz com que seu número total de avaliações cresça rapidamente conforme cresce o número de tentativas. Além disso, como um número aleatório é a cada tentativa, um mesmo conjunto poderá ser avaliado sucessivas vezes, caso o número gerado seja igual ou próximo a um gerado anteriormente.

O *Improved Probabilistic Batch Elimination* (IPBE) é uma solução baseada no PBE que elimina esses problemas. Para evitar retrabalho, o IPBE calcula uma única vez o fator de nocividade das otimizações (n avaliações) e o processo de tentativas é feito em seguida (c avaliações).

Outro aspecto fundamental é que o IPBE se baseia na ideia de que, se o PBE gera números aleatórios diferentes a cada tentativa e sempre um desses números traz o melhor resultado de c tentativas, então se $c \to \infty$ será possível encontrar um ponto no intervalo $[0, 1]$ que traz o melhor resultado que pode ser buscado pelo PBE.

Considerando tal ideia, o problema pode ser reduzido a uma simples busca por esse ponto. O PBE procura esse ponto gerando um número aleatório com c tentativas enquanto o IPBE busca maximizar as chances de encontrar tal número, fazendo com que pontos equidistantes no intervalo $[0, 1]$ sejam experimentados, de acordo com o número de tentativas. Portanto, sendo c o número de tentativas, a distância entre esses pontos é dada por:

$$intervalo = \frac{1}{c-1}$$

Algoritmo 7: *Improved Probabilistic Batch Elimination*

Dados: Programa (P); Conjunto Padrão (B); Número de tentativas (c)
Resultado: Melhor conjunto de otimizações encontrado (melhor_conj)

```
procedimento IPBE(P, B, c)
    tam_B ← tamanho(B);
    melhor_conj ← B;
    melhor_tempo ← execute(P, B);
    prob ← crie uma nova lista vazia;
    para i = 1 → tam_B faça
        Insira 0 ao final da lista prob;
    fim
    para i = 1 → tam_B faça
        Ligue todas as otimizações em B;
        Desligue a otimização i em B;
        tempo_i ← execute(P, B);
        se tempo_i > tempo_B então
            prob[i] ← tempo_i
        fim
    fim
    M ← maximo(prob);
    se M = 0 então
        retorne B;
    fim
    para i = 1 → tam_B faça
        prob[i] ← prob[i]/M;
    fim
    R ← 0;
    para j = 1 → c faça
        conj = B;
        intervalo ← 1/(c−1);
        para i = 1 → tam_B faça
            se conj[i] ≥ R então
                Desligue a otimização i em conj;
            fim
        fim
        tempo_t ← execute(P, conj);
        se tempo_t < melhor_tempo então
            melhor_tempo ← tempo_t;
            melhor_conj ← conj;
        fim
        R ← R + intervalo;
    fim
    retorne melhor_conj;
fim procedimento
```

Assim, quanto maior o número de tentativas, menor será a distância entre os pontos e, consequentemente, maior será o número de pontos espalhados no intervalo. O IPBE é descrito detalhadamente no Algoritmo 7.

Com as características apresentadas, o IPBE, além de distribuir igualmente os pontos de probabilidade no intervalo, possui complexidade $\Theta(c+n)$.

Dentre os algoritmos propostos e apresentados até agora, o IPBE apresenta a complexidade mais viável. No entanto, essa complexidade ainda é alta. No PSO cada programa precisa ser executado para que sua solução seja avaliada e, além disso, a execução do programa pode ser demorada. Assim, é necessário reduzir ao máximo o número de avaliações para que a abordagem seja viável em mais contextos.

As estratégias apresentadas anteriormente não utilizam nenhum tipo de conhecimento prévio para procurar conjuntos de otimizações. O uso de conhecimento prévio para seleção pode ser explorado para reduzir o espaço de busca do problema e, consequentemente, reduzir a complexidade do algoritmo de busca.

Outra abordagem proposta neste trabalho, o *CBR-Selector*, tem complexidade $\Theta(c)$, onde c é o número de conjuntos a serem avaliados de um espaço de busca previamente construído.

4.4 CBR-Selector

A abordagem apresentada nesta seção utiliza uma técnica de AM conhecida por *Case-Based Reasoning* (CBR) ou, Raciocínio Baseado em Casos (RBC). Sistemas RBC resolvem problemas novos baseando-se soluções anteriores, ou seja, algumas instâncias do problema são resolvidas como forma de treino e são posteriormente utilizadas como base para solução de novas (Aamodt e Plaza, 1994).

O *CBR-Selector* utiliza essa abordagem para prever bons conjuntos de otimizações para determinado programa, ou seja, para um programa que ainda não teve nenhuma solução melhor do que o conjunto padrão. O *CBR-Selector* foi projetado com uma arquitetura de três fases, que é apresentada na Figura 4.1.

Inicialmente, o Construtor do Espaço Exploratório (CEE) constroi um espaço de busca EE com informação sobre k programas (P_1, ..., P_k), que irão fazer parte do conhecimento prévio. Este espaço é estruturado da seguinte forma: $EE = \{(PC_1, [S_1, ..., S_n]);\ ...;\ (PC_k, [S_1, ..., S_n])\}$, onde PC_1 é o conjunto de *performance counters* (PCs) do programa P_1 e S_1 é o primeiro conjunto de otimizações considerado “bom” para este programa.

Os *Performance Counters* são informações dinâmicas da execução do programa e consistem de dados de desempenho tais como: número de instruções executadas, acessos

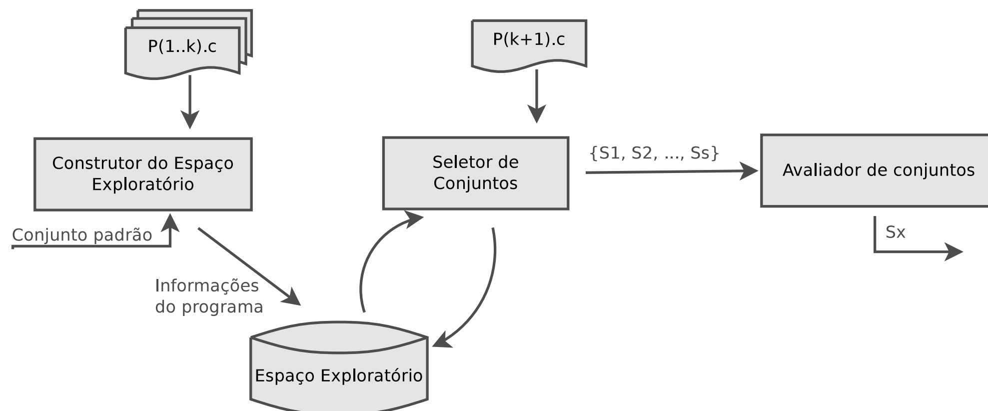


Figura 4.1: Arquitetura do *CBR-Selector*

a *cache*, falhas na *cache*, erros em predição de desvio, dentre outros. Estes dados são tradicionalmente usados para análise de desempenho de *hardware* (Dongarra et al., 2001), contudo servem para caracterizar o comportamento dinâmico de um programa (Cavazos et al., 2007).

Após a construção do espaço de busca EE, é possível selecionar transformações para programas para os quais ainda não se conhece solução.

Para o programa novo (P_{k+1}), o Seletor de Conjuntos coleta seus PCs (PC_{k+1}) e os compara com os PCs do espaço exploratório EE com a ajuda de um modelo de similaridade que identifica quais programas no espaço são similares ao programa novo. Baseado na similaridade, o Seletor de Conjuntos seleciona os melhores conjuntos de otimizações para P_{k+1} e eles são enviados para a próxima etapa: a avaliação. Nesta última etapa o Avaliador de Conjuntos avalia cada conjunto encontrado na etapa anterior com o objetivo de encontrar a melhor deles.

4.4.1 Construtor do Espaço Exploratório

O espaço exploratório (EE) armazena PCs e conjuntos de otimizações. O conjunto de PCs é coletado compilando e executando o programa P_i sem nenhuma otimização ativada. O objetivo em coletar PCs sem aplicar otimizações é que eles devem refletir o comportamento dinâmico do programa. Os conjuntos de otimizações são gerados de forma aleatória, de acordo com o Algoritmo 8. Assim, o espaço exploratório contém informação sobre k programas e, para cada um deles, um número variado de conjuntos de otimizações. O comportamento do método construtor é apresentado no Algoritmo 9.

Algoritmo 8: Gerador aleatório de conjuntos de otimizações

Dados: Tamanho mínimo (m) e tamanho máximo (M) do conjunto; Tabela de Otimizações (T)
Resultado: Um conjunto aleatório gerado (*conjunto_aleatorio*)

```
procedimento GERAR_CONJUNTO_ALEATORIO(m, M, T)
    conjunto_aleatorio ← crie uma lista vazia;
    r ← gere um número aleatório entre m e M;
    para i = 1 → r faça
        Selecione aleatoriamente uma otimização de T e adicione à lista
        conjunto_aleatorio;
    fim
    retorne conjunto_aleatorio;
fim procedimento
```

Algoritmo 9: Construtor do Espaço Exploratório

Dados: Programas (P[]); No. Conjuntos (R); Conjunto Padrão (B); Tabela de Otimizações (T)
Resultado: Espaço exploratório (EE)

```
procedimento CONSTRUTOR(P[], R, B, T[])
    b ← tamanho(B);
    EE ← crie uma nova lista vazia;
    para p ∈ P faça
        conjs ← crie uma nova lista vazia;
        Compile p sem nenhuma otimização, gerando o programa "P_O0";
        pc ← Colete os performance counters do programa "P_O0";
        Compile p com B, gerando o programa "P_base";
        t_base ← colete o tempo de execução para o programa "P_base";
        para j = 1 → R faça
            conj_aleatorio ← GERAR_CONJUNTO_ALEATORIO(0, b, T);
            Compile p com o conjunto conj_aleatorio, gerando o programa "P_j";
            t_j ← colete o tempo de execução para o programa "P_j";
            se t_j < t_base então
                Adicione o par ordenado (conj_aleatorio, t_j) ao final da lista conjs;
            fim
        fim
        se conjs não vazia então
            Ordene a lista conjs pelo valor de t_j em ordem não decrescente;
            Adicione o par ordenado (pc, conjs) ao final da lista EE;
        fim
    fim
    retorne EE;
fim procedimento
```

4.4.2 Seletor de Conjuntos

O objetivo deste mecanismo é identificar os melhores conjuntos do espaço exploratório, que são capazes de alcançar ganho de desempenho para um dado programa novo (P_{k+1}). Para alcançar este objetivo, o seletor compila P_{k+1} sem aplicar qualquer otimização, executa o código gerado e coleta seus PCs. Em seguida, um verificador estatístico de similaridade classifica cada programa do espaço exploratório, com base em sua similaridade com P_{k+1} e então extrai deles os melhores conjuntos.

Para medir a similaridade entre dois programas, o seletor fornece dois modelos:

Cosseno Neste modelo, a similaridade entre P_{k+1} e P_h é definida como:

$$sim(PC_{k+1}, PC_h) = \frac{\sum_{w=1}^{m} PC_{(k+1)w} \times PC_{hw}}{\sqrt{\sum_{w=1}^{m} (PC_{(k+1)w})^2} \times \sqrt{\sum_{w=1}^{m} (PC_{hw})^2}}$$

Jaccard Modificado Este modelo é uma adaptação do índice de similaridade de Jaccard para um espaço contínuo. Neste modelo, a similaridade entre P_{k+1} e P_h é definida como:

$$sim(PC_{k+1}, PC_h) = \frac{1}{m} \sum_{w=1}^{m} \frac{\min(PC_{(k+1)w}, PC_{hw})}{\max(PC_{(k+1)w}, PC_{hw})}$$

Nos dois modelos, m é o número de PCs e PC_{ab} é o PC b do programa a.

Estes modelos são baseados em coeficientes de similaridade utilizados para comparação de amostras estatísticas (Tan et al., 2005) e indicam que quanto maior a diferença entre os PCs, menor será o valor de similaridade. Além disso, se dois programas têm PCs com valores idênticos, $sim(PC_{k+1}, PC_x) = 1$.

Para comparar programas com esses modelos, é necessário normalizar os PCs, devido ao fato de diferentes tempos de execução gerarem diferentes valores para o mesmo PC. Portanto, os PCs são normalizados pelo número de instruções executadas. Assim, cada valor de PC será dado por:

$$PC_{normalized} = \{\frac{p}{TOT_INS}, p \in PC\}$$

onde PC é o conjunto de PCs e TOT_INS é o PC que representa o número de instruções concluídas.

Após determinar a similaridade entre cada programa do EE e o programa novo, o seletor irá extrair do EE os programas que fornecerão conjuntos de otimização. Para esta extração o seletor pode utilizar diferentes abordagens, a saber:

Probabilística Nesta estratégia, um número no intervalo $(0, 1]$ é gerado aleatoriamente, e o verificador de similaridade irá utilizá-lo para selecionar programas cuja similaridade é maior ou igual a ele. Esta etapa gera o espaço EE'.

Justa Nesta estratégia, cada programa no espaço EE é um candidato potencial para fornecer bons conjuntos. Neste caso $EE' = EE$.

Após selecionar quais programas irão fornecer os conjuntos, é necessário selecionar quais conjuntos são potenciais soluções. Nesta etapa, cada programa $P_h \in EE'$ irá contribuir com L_h conjuntos, tal que L_h é definida como:

$$L_h = \left\| L \times \frac{sim(PC_{k+1}, PC_h)}{\sum_{x=1}^{k} sim(PC_{k+1}, PC_x)} \right\|$$

onde L é um parâmetro do sistema que indica o número de conjuntos que devem ser avaliados, PC_a é o conjunto de PCs do programa a e sim é similaridade entre dois programas.

Quando o seletor finaliza sua tarefa, envia L conjuntos de otimizações para a próxima etapa, que fará a validação destas. O Algoritmo 10 apresenta os passos da seleção.

4.4.3 Avaliador de Conjuntos

O processo de validação verifica qual conjunto é o melhor entre os L selecionados, compilando P_{k+1} com cada um deles, executando-o e comparando o desempenho de cada um dos códigos gerados em relação ao conjunto padrão. O Algoritmo 11 apresenta esse processo.

Algoritmo 10: Seletor de Conjuntos

Dados: Programa novo (P); No. de Conjuntos (L); Espaço Exploratório (EE);
Resultado: Conjuntos preditos para P ($conjs$)
procedimento SELETOR(P, L, EE)
  $conjs \leftarrow$ crie uma nova lista vazia;
  Compile P sem nenhuma otimização, gerando o programa "P_O0";
  $pc_n \leftarrow$ Colete e normalize os *performance counters* do programa "P_O0";
  $tam_{EE} \leftarrow$ tamanho(EE);
  $lista_probs \leftarrow$ crie uma nova lista vazia;
  para $i = 1 \rightarrow tam_{EE}$ **faça**
    $p \leftarrow$ meça a similaridade entre pc_n e os *performance counters* do programa $EE[i]$;
    Insira p ao final da lista $lista_probs$;
  fim
  $EE' \leftarrow$ crie uma nova lista vazia;
  se *Seleção Probabilística* **então**
    $r \leftarrow$ gere um número aleatório entre 0 e 1;
    para $i = 1 \rightarrow tam_{EE}$ **faça**
      se $lista_probs[i] \geq r$ **então**
        Insira $EE[i]$ ao final da lista EE';
      fim
      senão
        Insira $null$ ao final da lista EE';
        $lista_probs[i] \leftarrow 0$;
      fim
    fim
  fim
  se *Seleção Justa* **então**
    $EE' = EE$;
  fim
  $soma_probs \leftarrow$ some os elementos da lista $lista_probs$;
  para $i = 1 \rightarrow tam_{EE}$ **faça**
    $L_i = \left\| L \times \frac{lista_probs[i]}{soma_probs} \right\|$
    Adicione os primeiros L_i conjuntos de $EE'[i]$ ao final da lista $conjs$;
  fim
  retorne $conjs$;
fim procedimento

Algoritmo 11: Avaliador de Conjuntos

Dados: Programa novo (P); Conjunto Padrão (B); Conjuntos do seletor (S)
Resultado: Melhor conjunto para P ($conj_{melhor}$)
procedimento AVALIADOR(P, B, S)
  Compile P com o conjunto padrão, gerando o programa "P_base";
  $t_{base} \leftarrow$ colete o tempo de execução do programa "P_base";
  $t_{melhor} \leftarrow t_{base}$;
  $conj_{melhor} \leftarrow B$;
  $L \leftarrow tamanho(S)$;
  para $s \in S$ **faça**
    Compile P com o conjunto s, gerando o programa "P_s";
    $t_s \leftarrow$ colete o tempo de execução do programa "P_s";
    se $t_s < t_{melhor}$ **então**
      $conj_{melhor} \leftarrow s$;
    fim
  fim
  retorne $conj_{melhor}$;
fim procedimento

É importante observar que a etapa de validação é uma estratégia conservadora porque se não houver um conjunto melhor do que o conjunto padrão, o Avaliador de Conjuntos o retornará como o melhor conjunto.

4.5 Considerações Gerais

Este capítulo apresentou as abordagens propostas no presente trabalho para mitigação do PSO. Foram propostas e implementadas quatro abordagens. Para verificar a eficiência e viabilidade de cada abordagem, deve ser feita uma experimentação com diferentes programas. O capítulo seguinte apresenta a avaliação experimental de cada uma das quatro abordagens, destacando os principais pontos e comparando os resultados entre elas. Além disso, é feita uma comparação com os algoritmos BE, IE e CE apresentados na Seção 3.3.

5

Avaliação Experimental

Este capítulo apresenta os experimentos realizados para a avaliação de cada uma das abordagens expostas no capítulo anterior. O principal objetivo é avaliar a viabilidade das abordagens desenvolvidas, mediante a apresentação de resultados experimentais de cada uma delas, bem como comparação entre estes, considerando a qualidade das soluções e o tempo que elas demoraram para ser encontradas.

5.1 Metodologia

Plataforma Todos os experimentos foram conduzidos com o uso da LLVM versão 3.1 (Lattner e Adve, 2004). O equipamento utilizado foi uma máquina Intel x86_64, com processador Core I7-2600, executando a 3.4GHz com *caches* de instruções e de dados L1, L2, L3 e RAM de 32K, 256K, 8M e 4GB, respectivamente. O sistema operacional utilizado foi o Debian, com kernel 2.6.32-5-amd64. Esta máquina possibilita a análise de 48 *performance counters* (PCs), utilizados em uma das abordagens propostas. Os PCs foram coletados com o uso da ferramenta *Perfsuite* (Kufrin, 2005) juntamente com a biblioteca PAPI (Dongarra et al., 2001).

Benchmarks Para a experimentação de todas as estratégias foi utilizado o conjunto de *benchmarks MiBench*.

Especificamente para a experimentação do *CBR-Selector*, foram utilizados, além do *MiBench*, *benchmarks* dos conjuntos *SNU NPB Serial* [1], *Shootout* [2] e *SPLASH-2*

[1] http://aces.snu.ac.kr/Center_for_Manycore_Programming/SNU_NPB_Suite.html
[2] http://benchmarksgame.alioth.debian.org

Serial (Woo et al., 1995). No *CBR-Selector*, existe a necessidade de geração de um espaço de busca inicial. Portanto, foram separados conjuntos de treino e teste, sendo que o conjunto de treino foi utilizado exclusivamente para a construção do espaço e o conjunto de teste para a experimentação da abordagem proposta. Para treino, 31 programas foram usados: todos do conjunto *MiBench* (21 programas), 6 do conjunto *SNU NPB Serial* e 4 do conjunto *Shootout*. A experimentação do *CBR-Selector* também contemplou dois tipos de avaliação: uma *leave-one-out cross validation* entre todos os 31 programas citados acima e também uma treino e teste, que utiliza os 31 programas como conjunto de treino e 25 programas como conjunto de teste: todos do *SPLASH-2 Serial* (9 programas), 3 do *SNU NPB Serial* e 13 do *Shootout*.

Conjunto Padrão O nível de otimização `-O3` foi utilizado como conjunto padrão em todos os experimentos. Na LLVM, a opção `-O3` aplica as seguintes otimizações, em ordem:

1. -targetlibinfo
2. -no-aa
3. -tbaa
4. -basicaa
5. -globalopt
6. -ipsccp
7. -deadargelim
8. -instcombine
9. -simplifycfg
10. -basiccg
11. -prune-eh
12. -inline
13. -functionattrs
14. -argpromotion
15. -sroa
16. -domtree
17. -early-cse
18. -simplify-libcalls
19. -lazy-value-info
20. -jump-threading
21. -correlated-propagation
22. -simplifycfg
23. -instcombine
24. -tailcallelim
25. -simplifycfg
26. -reassociate
27. -domtree
28. -loops
29. -loop-simplify
30. -lcssa
31. -loop-rotate
32. -licm
33. -lcssa
34. -loop-unswitch
35. -instcombine

36. -scalar-evolution
37. -loop-simplify
38. -lcssa
39. -indvars
40. -loop-idiom
41. -loop-deletion
42. -loop-unroll
43. -memdep
44. -gvn
45. -memdep
46. -memcpyopt
47. -sccp
48. -instcombine
49. -lazy-value-info
50. -jump-threading
51. -correlated-propagation
52. -domtree
53. -memdep
54. -dse
55. -adce
56. -simplifycfg
57. -instcombine
58. -strip-dead-prototypes
59. -globaldce
60. -constmerge
61. -preverify
62. -domtree
63. -verify

Objetivo O objetivo de cada estratégia apresentada é encontrar um conjunto de otimizações que supere o desempenho do nível de otimização `-O3`, em termos de tempo de execução.

Validação A validação dos resultados é baseada na média de dez execuções para cada instância.

Medida de Desempenho O desempenho de cada abordagem é baseado no *speedup* alcançado, sendo este determinado como segue:

$$Speedup_p = \frac{Tempo_{O3}}{Tempo_p}$$

O ganho de desempenho é dado por:

$$Ganho_p = (Speedup_p - 1) \times 100\%$$

5.2 Variable Neighborhood Search

As metaheurísticas são técnicas com algoritmos aproximativos que tentam combinar métodos heurísticos básicos para fazer uma busca eficiente em espaços muito grandes (Blum e Roli, 2003). Tais técnicas são utilizadas para procurar soluções para problemas de otimização conhecidos, porém de uma forma genérica.

Como cada problema tem suas peculiaridades, nem toda metaheurística encontrará boas soluções para determinado problema, sendo necessária uma experimentação para verificar a aplicabilidade da metaheurística ao problema.

Como o PSO é um problema que tem um espaço de busca combinatório, é relevante avaliar se uma metaheurística genérica encontra soluções para ele.

Tendo isso em vista, o principal objetivo desta seção é avaliar o desempenho da metaheurística VNS aplicada ao PSO.

A estratégia apresentada no Algoritmo 5 foi aplicada para cada programa do *MiBench*. Os parâmetros específicos deste algoritmo foram definidos como:

- Número de iterações: 30 e;
- Solução inicial: o conjunto padrão.

5.2.1 Speedups

É importante citar inicialmente que a VNS alcançou *speedups*, em média, maiores do que todas as outras abordagens. Além de um resultado qualitativo, ou seja, um *speedup* médio significativamente alto, esta abordagem apresentou um resultado quantitativo peculiar: houve ganho de desempenho para todos os programas experimentados. O ganho de desempenho médio foi de 16,9%, sendo o menor 0,6% (`lame`) e o maior 65% (`susanc`). A maioria dos programas apresentou um ganho maior que 10%. A Figura 5.1 apresenta o *speedup* alcançado pela VNS para cada programa do *MiBench*.

Tais resultados evidenciam que a abordagem proposta neste trabalho com uso de VNS é promissora, pois além de alcançar *speedups* relativamente bons, consegue fazer isso para a maioria dos programas. Conforme já mencionado, a forma de exploração de espaço de busca da VNS inclui a seleção de quais otimizações devem compor o conjunto, a ordem em que devem ser aplicadas e ainda considera a possibilidade de repetir a aplicação de uma ou mais otimizações. Nenhuma das outras estratégias exploram o espaço de busca de forma tão completa quanto esta. Provavelmente tais características são responsáveis por

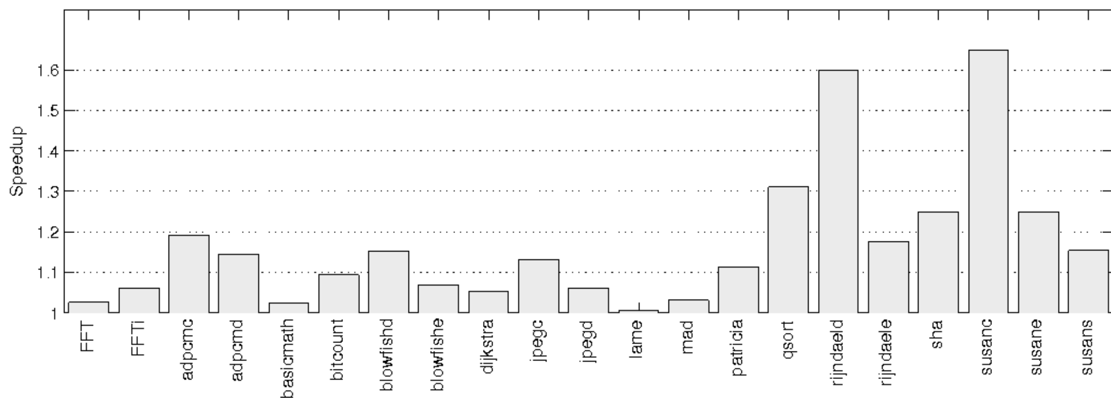


Figura 5.1: *Speedups* alcançados pela metaheurística VNS

fazer com que esta abordagem alcance os maiores e mais amplos resultados, em termos de *speedup*.

5.2.2 Número de Avaliações

Apesar da qualidade dos conjuntos encontrados, o tempo que esta abordagem leva para encontrá-los não é muito favorável. Seu número médio de avaliações foi de 1835, sendo que o resultado mais rápido (`dijkstra`) necessitou de 195 avaliações e o mais demorado (`qsort`) chegou a 5148 avaliações. Ou seja, para encontrar uma boa sequência para um programa que compila e executa em 1 segundo a VNS levou, em média, pouco mais que 30 minutos. Para um caso como este talvez não seja relevante a questão do tempo, porém, se forem considerados programas com execução morosa, tal fator passa a ser bem relevante, tornando seu uso inviável. O número de avaliações que a VNS executou para cada *benchmark* neste experimento pode ser visto na Figura 5.2. Para uma melhor visualização de qualidade de resultado *versus* tempo de resposta, a Figura 5.3 apresenta a relação entre o número de avaliações e o *speedup* em um mesmo plano, onde cada ponto é um dos programas avaliados.

Os pontos da Figura 5.3 apresentam um certo crescimento no *speedup* em função do crescimento do número de avaliações. Apesar de a regra ser "quanto maior o número de avaliações melhor será o resultado", isso nem sempre ocorrerá com as instâncias do PSO, havendo inúmeras exceções. As diferentes características dos programas tornam alguns mais fáceis de se obter *speedup* enquanto para outros há uma dificuldade maior. Isso pode ser observado claramente nos dois maiores *speedups*: ambos foram obtidos com um número de avaliações muito pequeno (254 e 368), relativo às outras instâncias. Já o

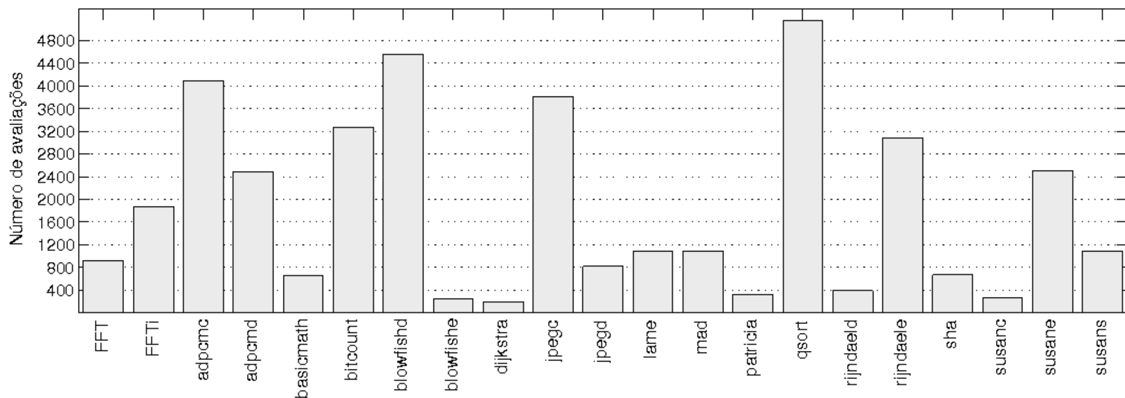


Figura 5.2: Número de avaliações para as soluções encontradas pela VNS

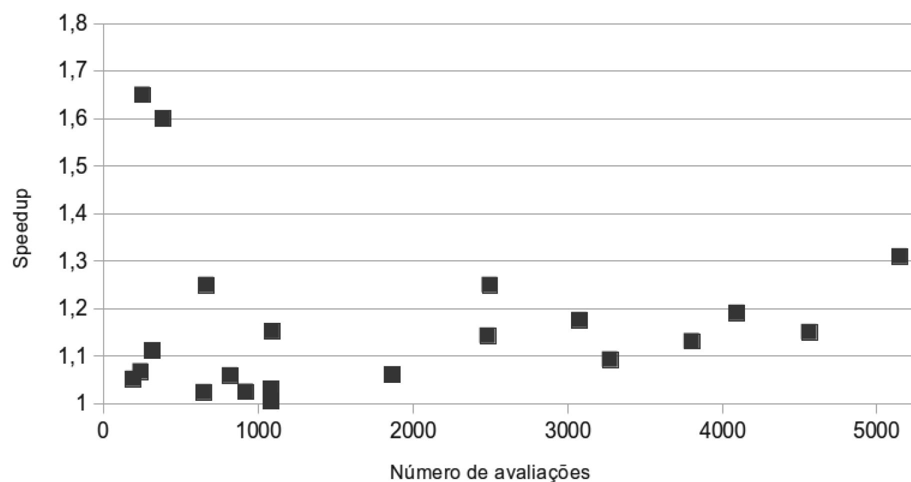


Figura 5.3: *Speedups* e número de avaliações da VNS

menor *speedup* no conjunto de *benchmarks* necessitou de mais de 1000 avaliações para ser alcançado.

Em essência, os dados apresentados indicam que, apesar de a VNS alcançar os *speedups* mais altos, seu tempo de resposta é muito lento para ser aplicado ao PSO. Apesar de alguns programas apresentarem um bom *speedup* com relativamente poucas avaliações, como `susanc` (1,65/254), `rijndaeld` (1,6/386) e `sha` (1,25/667), a maioria dos casos levou mais do que 1000 para alcançar os *speedups* apresentados. Isto pode ser até aceitável em aplicações bem rápidas, como é o caso dos *benchmarks* do *MiBench* experimentados aqui. Mas programas mais demorados podem ser inviáveis de serem processados com este algoritmo.

Apesar disso, o fato dos dois maiores *speedups* terem sido alcançados com poucas avaliações sugere que algumas instâncias do PSO podem ser favorecidas, no quesito qualidade *versus* tempo de resposta, se resolvidas com VNS. Como isso não ocorre sempre,

é necessário avaliar outras estratégias de busca para o PSO. Além disso, mesmo o menor número de avaliações da VNS, que é o caso do `dijkstra` (195), pode ser considerado ainda um número alto em alguns casos.

5.3 Probabilistic Batch Elimination

O *Probabilistic Batch Elimination* (PBE) é um algoritmo escrito a partir da alteração do BE, proposto por Pan e Eigenmann (2006). Um diferencial entre o PBE e as abordagens apresentadas por Pan e Eigenmann (2006) é o uso de uma estratégia probabilística para eliminação de OPNs.

No experimento executado, o conjunto padrão teve tamanho 63 e o número de tentativas utilizado foi de $c = 1, 5, 10, 20$ e 30. Assim, o número de avaliações foi exatamente 63, 315, 630, 1260 e 1890, respectivamente.

Diferente do BE, o PBE não retira todas as otimizações que, isoladamente, afetam negativamente o código. As eliminações são feitas com base em uma probabilidade gerada aleatoriamente. É esperado que tal estratégia forneça mais oportunidades para busca de diferentes soluções para o PSO, o que provavelmente resultará em um maior ganho de desempenho que o do BE.

5.3.1 Speedups

A Figura 5.4 apresenta o *speedup* alcançado pelo PBE para cada um dos programas do *MiBench* para cada número de tentativas.

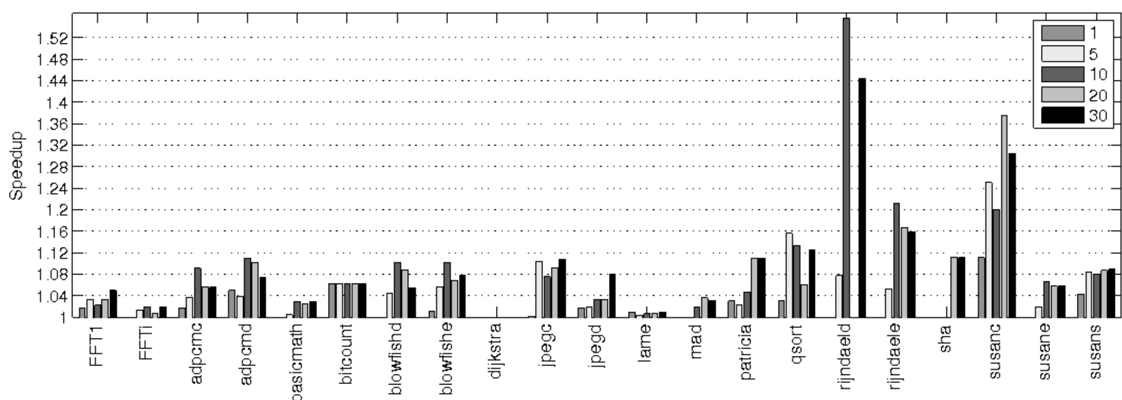


Figura 5.4: *Speedups* alcançados pelo PBE

É interessante observar o comportamento aleatório do algoritmo. Em vários casos, o maior *speedup* do programa não ocorreu com o maior número de tentativas e sim com

um número inferior. É o caso dos programas adpcmc, adpcmd, blowfishd, blowfishe, mad, qsort, rijndaeld, rijndaele, susanc e susane. Como o comportamento do PBE é baseado no número aleatório gerado, um número "melhor" pode ter sido gerado em dez avaliações do que em outras 30, por exemplo. No entanto, quanto maior o número de avaliações menos casos destes devem ocorrer. Se forem observadas as execuções com uma tentativa, nenhuma foi melhor do que com 30. Nas execuções com cinco tentativas, apenas o qsort apresentou melhores resultados do que com 30 tentativas.

Essa característica permite que o PBE seja uma estratégia utilizada de forma parametrizada, possibilitando a geração de uma melhor solução quando se dispõe de mais tempo e também a geração de uma solução de menor qualidade quando o tempo for um fator crítico.

Outro ponto a ser destacado é a qualidade do resultado encontrado para os programas rijndaeld e susanc. Tais instâncias também obtiveram bons resultados em outras abordagens, reforçando a hipótese de que alguns programas são mais propícios à otimização que outros. O rijndaeld foi o segundo melhor resultado da VNS e o primeiro do PBE. Já o susanc foi o melhor resultado da VNS e o segundo melhor do PBE. Analogamente, o lame permaneceu sem bons resultados, assim como ocorreu na VNS.

Os resultados do PBE, apesar de ainda um tanto distantes dos alcançados pela metaheurística VNS, foram significativos: seu ganho de desempenho médio chegou a 9,76% contra 16,9% da VNS. Além disso, o PBE alcançou ganho de desempenho para todos os programas, com exceção do dijkstra.

Neste contexto o PBE pode ser uma alternativa à VNS já que, a despeito da qualidade inferior dos resultados, pode fornecer soluções em um tempo menor.

5.3.2 Probabilidades e Speedups

É fundamental analisar a relação entre as probabilidades do PBE e os *speedups* para os programas, com o objetivo de verificar em que faixa de probabilidades ocorrem os melhores resultados. Tal relação pode ser visualizada na Figura 5.5, onde cada símbolo representa o *speedup* de um programa com seu respectivo número de tentativas, que pode ser visto na legenda do canto superior direito.

É interessante notar que em poucos casos se alcançou o melhor desempenho nos extremos, ou seja, com probabilidades 1,0 e 0,0. É também importante observar que o algoritmo BE clássico é equivalente ao PBE executando com probabilidade 0,0 e que executar o PBE com probabilidade 1,0 é equivalente a eliminar apenas a otimização mais nociva, como faz o IE a cada iteração.

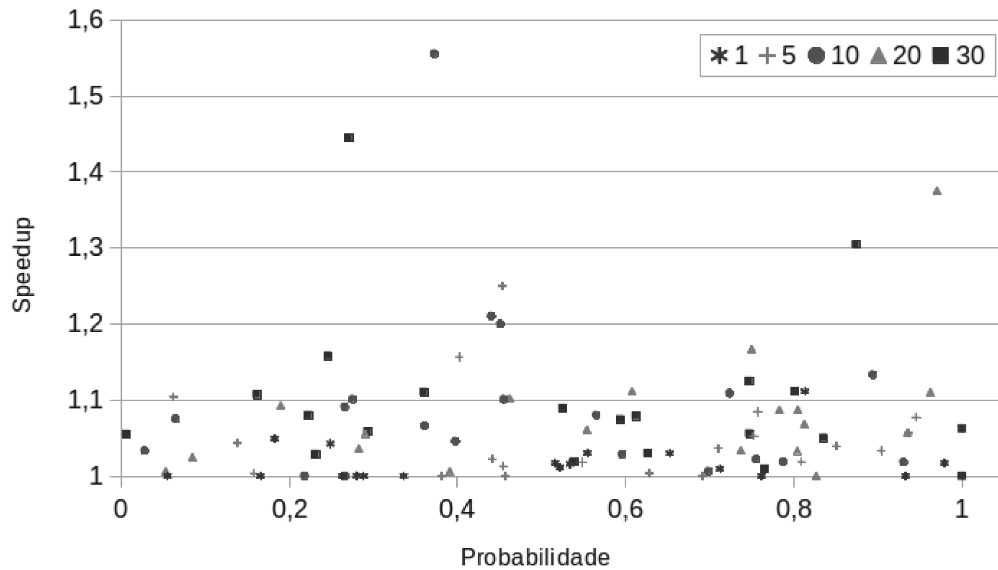


Figura 5.5: Probabilidades e *speedups* do PBE

Os dados do gráfico mostram claramente que a abordagem probabilística, diferente da BE clássica que se limita ao ponto 0,0, alcança muitos *speedups* em outros pontos.

Dada a heterogeneidade dos programas, não houve um agrupamento em apenas uma região de pontos. No entanto, se o eixo das probabilidades for seccionado em dez partes de 0,1, será possível notar que a maior concentração ocorre nos intervalos $(0,7; 0,8]$ e $(0,9; 1,0]$, onde há 13 programas com *speedup* em cada intervalo. Logo em seguida, os intervalos $(0,2; 0,3]$ e $(0,5; 0,6]$ também apresentam um alto agrupamento, com 11 programas com *speedup* em cada um deles.

5.3.3 Número de Avaliações

Um ponto positivo do PBE que já foi destacado é o fato de ele permitir a configuração do número de avaliações. Na Tabela 5.1 pode ser visualizada a evolução da qualidade dos resultados em função do número de avaliações executadas.

Tentativas	Avaliações	Ganho médio
1	63	1,89%
5	315	5,11%
10	630	9,33%
20	1260	7,49%
30	1890	9,76%

Tabela 5.1: Ganho de desempenho médio para cada número de tentativas do PBE

Os dados da Tabela 5.1 sugerem que o PBE não necessita de um número de tentativas tão alto para alcançar um bom desempenho. Um resultado satisfatório, 9,33% de ganho, foi alcançado com apenas dez tentativas (630 avaliações).

Outra estratégia foi proposta visando reduzir ainda mais o número de avaliações na busca de soluções para o PSO. Uma alteração do PBE, apresentada na seção 4.3, é avaliada na seção seguinte.

5.4 Improved Probabilistic Batch Elimination

Nesta seção é apresentada a experimentação do algoritmo *Improved Probabilistic Batch Elimination* (IPBE), derivado do PBE. A proposta principal do IPBE é procurar conjuntos de forma semelhante ao PBE, porém procurando reduzir o número de avaliações. Isso é alcançado evitando o retrabalho que ocorre em cada uma das tentativas do PBE. Enquanto o PBE, a cada tentativa, calcula a probabilidade de cada otimização ser retirada do conjunto padrão, o IPBE faz isso apenas uma vez e reutiliza tal informação para as demais tentativas. Por isso, enquanto o PBE tem complexidade $\Theta(cn)$, o IPBE tem complexidade $\Theta(c+n)$. Além disso, como mencionado na Seção 4.3, o IPBE seleciona pontos no intervalo $[0,0; 1,0]$, segmentando-o. Assim, o número mínimo de tentativas deve ser três, para que o intervalo seja dividido em dois segmentos. Portanto, neste experimento o número de tentativas foi $c = 3, 5, 10, 20$ e 30 e, como o tamanho do conjunto padrão foi de 63, o número de avaliações foram de $66, 68, 73, 83$ e 93, respectivamente.

Com tal configuração, o principal objetivo deste experimento é verificar se é possível, por meio destas modificações no PBE, reduzir seu número de avaliações, porém sem dispensar a qualidade das soluções.

5.4.1 Speedups

A Figura 5.6 apresenta os *speedups* alcançados pelo IPBE para cada número de avaliações.

O primeiro ponto a ser destacado é a qualidade das soluções encontradas, que não reduziu em relação ao PBE. Ao invés disso, o ganho de desempenho foi superior, chegando a uma média de 10,02% com 30 tentativas, o equivalente a apenas 93 avaliações. Também deve ser notada a semelhança dos resultados do IPBE com os do PBE, como por exemplo os maiores *speedups*, alcançados nos programas `rijndaeld` e `susanc` assim como também os piores, `dijkstra` e `lame`. Estes resultados favorecem a hipótese da construção do IPBE de que é possível evitar recalcular a probabilidade de cada otimização ser retirada do conjunto padrão.

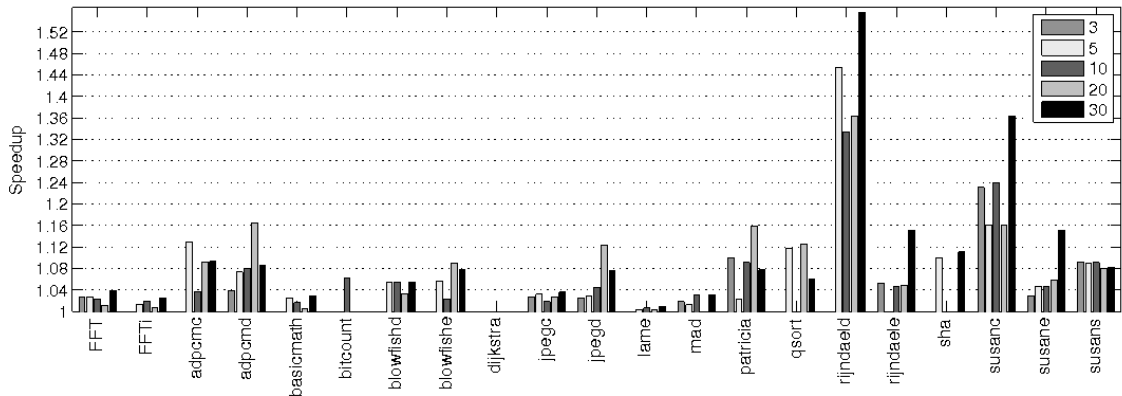


Figura 5.6: *Speedups* alcançados pelo IPBE

Além disso, é importante observar uma característica diferenciada dos dados da Figura 5.6. É notável que não é na maioria dos casos que a execução com 30 tentativas trouxe o melhor resultado. Uma característica particular do IPBE pode ter sido responsável por essa diferença. Como o IPBE segmenta o intervalo $[0,0;1,0]$ de forma que o número de pontos equidistantes no intervalo seja igual ao número de tentativas, pode haver pontos contemplados em três tentativas enquanto em dez tentativas eles não são contemplados. Por exemplo, em três tentativas, os pontos contemplados serão: $0,0;0,5;1,0$ e em dez tentativas os pontos contemplados serão: $0,0;\frac{1}{9};\frac{2}{9};\frac{1}{3};...;1,0$. Mesmo experimentando mais pontos, as dez tentativas não atingirão o ponto 0,5, atingido pelas três tentativas. Portanto, pode haver pontos mais favorecidos que serão alcançados de acordo com a segmentação do intervalo, o que é determinado diretamente pelo número de tentativas. Esses resultados também fortalecem a ideia fundamental do IPBE, que é a de reduzir o PSO à busca de uma probabilidade de eliminação de otimizações ótima.

5.4.2 Probabilidades e Speedups

Assim como no PBE, é interessante analisar a relação entre as probabilidades e os *speedups* alcançados pelo IPBE. A Figura 5.7 apresenta tal relação.

Uma análise mais detalhada permite concluir que há intervalos mais favorecidos na Figura 5.7. Da mesma forma como exposto na análise do PBE, se o eixo das probabilidades for seccionado em 10 intervalos de tamanho 0,1, alguns intervalos apresentarão uma quantidade significativamente superior a outros.

O intervalo que apresentou o maior número de *speedups* (18) foi $(0,9;1,0]$, o que não ocorreu no PBE, em que o mesmo intervalo não foi tão favorecido quanto aqui. Assim, no IPBE ocorreram muitos *speedups* até mesmo com probabilidade $1,0$. Esse

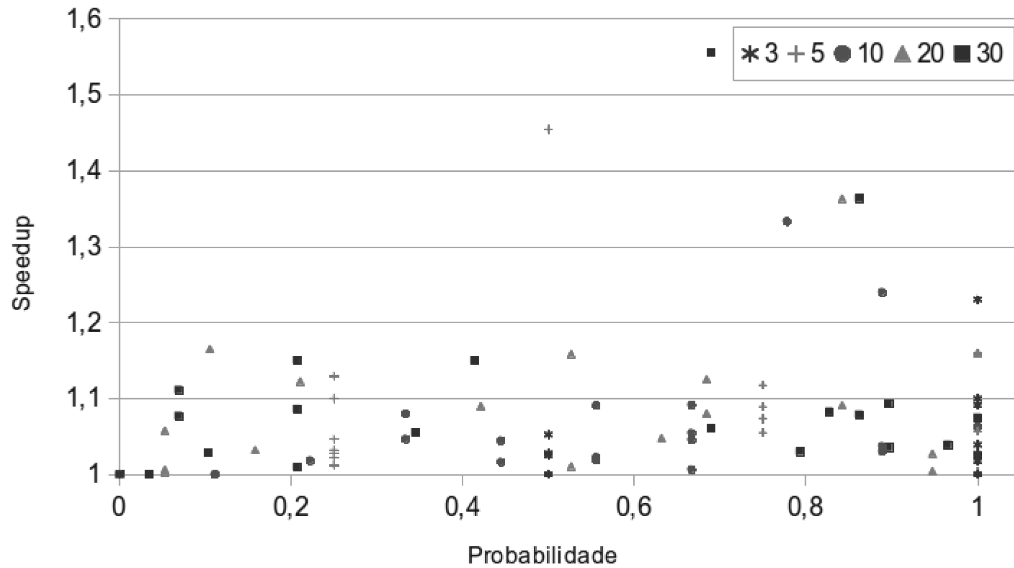


Figura 5.7: Probabilidades e *speedups* do IPBE

comportamento se deve à diferença entre o PBE e o IPBE no que tange à escolha das probabilidades. Enquanto o PBE escolhe a probabilidade aleatoriamente, o IPBE sempre secciona o intervalo $[0,0; 1,0]$, o que favorece maior experimentação com probabilidade igual a $0,0$ e a $1,0$.

Além desse caso particular, os intervalos $(0,2; 0,3]$, $(0,6; 0,7]$ e $(0,8; 0,9]$ se destacam, com número de *speedups* de 13, 9 e 10, respectivamente. É relevante observar que os intervalos $(0,2; 0,3]$ e $(0,9; 1,0]$ são destaques tanto no PBE quanto no IPBE. E também há intervalos de destaque muito próximos nas duas abordagens como os $(0,6; 0,7]$ e $(0,8; 0,9]$ no IPBE, que estão próximos do $(0,7; 0,8]$ do PBE.

Estas semelhanças validam a abordagem do PBE/IPBE, que restringe o problema à busca de uma probabilidade de eliminação das otimizações nocivas. Como um mesmo conjunto de *benchmarks* apresentou distribuições de resultados semelhantes, é provável que para um determinado grupo de programas seja viável utilizar um valor fixo de probabilidade, o que reduziria os custos das buscas dos conjuntos de otimizações. No entanto, uma experimentação mais ampla deve ser feita para validação de tal hipótese.

5.4.3 Número de Avaliações

Como o IPBE também permite a parametrização do número de avaliações, é essencial analisar como a qualidade do resultado cresce quando cresce o número de avaliações. A Tabela 5.1 mostra o ganho de desempenho médio do IPBE para cada número de tentativas.

Tentativas	Avaliações	Ganho médio
3	66	3,04%
5	68	6,87%
10	73	6,00%
20	83	7,36%
30	93	10,02%

Tabela 5.2: Ganho de desempenho médio para cada número de tentativas do IPBE

Os dados da Tabela 5.2 indicam que o IPBE é uma estratégia melhor que o PBE, pelo fato do IPBE alcançar um desempenho superior com um número menor de avaliações. O maior destaque do IPBE, no entanto, não foi somente o *speedup* mas sim a eficiência na busca de tais soluções de qualidade. Seu ganho de desempenho de 10,02% foi alcançado com apenas 93 avaliações, ao passo que o da VNS esteve em uma média de 1835.

Assim, em contextos onde o tempo de resposta é crucial, o IPBE é uma estratégia eficiente ante as demais apresentadas, por pelo menos três motivos:

- A característica parametrizável do algoritmo;
- Busca de soluções com qualidade satisfatória;
- Complexidade menor que as demais estratégias ($\Theta(n + c)$).

Apesar dos melhores resultados alcançados pelo IPBE, o número de avaliações necessárias para chegar a tal solução ainda é considerável.

No cenário ideal existiria um algoritmo com complexidade $\Theta(1)$ que encontraria a melhor solução. Apesar de um algoritmo como tal não ser conhecido, é possível projetar um algoritmo com complexidade $\Theta(c)$, onde c é uma constante definida pelo usuário, que procure por bons conjuntos de otimizações, isto é, um algoritmo que não tenha sua complexidade dependente do número de otimizações no conjunto padrão. Esta é a proposta do algoritmo experimentado na seção seguinte.

5.5 CBR-Selector

Nesta seção é apresentada a avaliação da abordagem denominada *CBR-Selector* que, diferente de todas as abordagens apresentadas anteriormente, utiliza conhecimento prévio para solucionar o PSO. Enquanto os algoritmos anteriores apresentam complexidade dependente do número de otimizações no conjunto padrão, a abordagem aqui apresentada apresenta complexidade $\Theta(c)$, em que c é o número de conjuntos de otimizações a serem

avaliados do espaço exploratório inicial, ou seja, o número de avaliações não dependerá diretamente do tamanho do conjunto padrão. Os detalhes da implementação da solução *CBR-Selector* podem ser vistos na seção 4.4.

O principal objetivo da experimentação aqui apresentada é mostrar que o uso de conhecimento prévio é conveniente na busca de soluções para o PSO. A construção do espaço exploratório e as estratégias de seleção utilizadas pelo *CBR-Selector* são os dois aspectos principais que devem ser avaliados. O que se espera da abordagem proposta é que esta seja capaz de encontrar soluções com qualidade igual ou superior às das soluções dos algoritmos anteriores, mas considerando a questão crucial do tempo de resposta. Enquanto as abordagens anteriores necessitaram de no mínimo 93 avaliações para encontrar as soluções, a estratégia que será experimentada aqui necessitou de apenas 20 avaliações.

5.5.1 Configurações Específicas

Como esta abordagem é diferenciada das outras, algumas configurações específicas são necessárias.

Parâmetros do *CBR-Selector* Alguns parâmetros precisam ser escolhidos para execução do *CBR-Selector*. Primeiramente, é preciso definir quantos conjuntos devem ser gerados aleatoriamente para cada programa na construção do espaço exploratório (R). Além disso, é necessário definir quantos conjuntos serão avaliados pelo mecanismo seletor (L). Neste experimento, foram tomados $R \leftarrow 500$ e $L \leftarrow 20$.

Tabela de Otimizações Os experimentos do *CBR-Selector* utilizam 51 otimizações disponíveis na versão 3.1 da LLVM. Essas otimizações são apresentadas na Tabela 5.3.

O Espaço Exploratório Para ser suficientemente representativo, o espaço exploratório deve ser construído com *benchmarks* que tenham comportamento bem diversificado. Isso se deve ao fato do Seletor de Conjuntos executar suas escolhas com base em similaridade estatística. Quanto mais diversificado for o comportamento dos *benchmarks*, maior será a probabilidade do seletor encontrar um programa similar no espaço exploratório para determinado programa novo.

Neste experimento, quatro grupos de *benchmarks* foram escolhidos, a saber: o *MiBench*, o *SNU NPB Serial*, o *Shootout* e o *SPLASH-2*.

A dificuldade potencial em resolver o PSO é que seu espaço de busca é muito grande e o processo de enumerá-lo completamente é impraticável para conjuntos com

Otimizações		
-adce	-always-inline	-argpromotion
-bb-vectorize	-block-placement	-break-crit-edges
-codegenprepare	-constmerge	-constprop
-dce	-deadargelim	-die
-functionattrs	-globaldce	-globalopt
-gvn	-indvars	-inline
-instcombine	-instsimplify	-intervals
-ipconstprop	-ipsccp	-jump-threading
-licm	-loop-deletion	-loop-instsimplify
-loop-reduce	-loop-rotate	-loop-simplify
-loop-unroll	-loop-unswitch	-loweratomic
-lowerinvoke	-lowerswitch	-mem2reg
-memcpyopt	-mergefunc	-mergereturn
-partial-inliner	-partial-inliner	-reassociate
-regions	-scalarrepl	-sccp
-simplify-libcalls	-simplifycfg	-sink
-strip	-strip-dead-prototypes	-tailcallelim

Tabela 5.3: Otimizações da LLVM utilizadas pelo *CBR-Selector*

dezenas de otimizações, já que o tamanho desse espaço cresce exponencialmente em função do número de otimizações. Nos experimentos realizados para avaliação do *CBR-Selector* foram utilizadas 51 otimizações e, neste caso, o número de conjuntos do espaço exploratório é de 51^{51}, se for considerada a ordem de aplicação das otimizações, sem repetição.

No entanto, como o *CBR-Selector* utiliza informações dinâmicas de vários programas, esse número pode ser reduzido significativamente. A proposta desta abordagem é que um pequeno espaço exploratório seja um espaço de busca útil para vários outros programas para os quais ainda não foram encontrados bons conjuntos de otimizações. Para que isso seja possível, uma estratégia para construir um espaço exploratório reduzido eficientemente é necessária. A estratégia adotada aqui foi utilizar *benchmarks* com um curto tempo de execução e que representam várias classes de programas.

Foram utilizados 31 programas para construção do espaço exploratório. Inicialmente, o Construtor do Espaço Exploratório gerou $500 \times 30 = 15000$ conjuntos (na experimentação *leave-one-out*) ou $500 \times 31 = 15500$ conjuntos (na experimentação treino e teste), um número que representa uma fração muito pequena do espaço de busca total. Tais conjuntos foram filtrados pelo construtor e apenas os melhores

que o conjunto padrão foram utilizadas pelo Seletor de Conjuntos. O número de conjuntos para cada programa após esse filtro é apresentado na Tabela 5.4, totalizando 3804.

Benchmark	#	**Benchmark**	#	**Benchmark**	#
CG	0	MG	4	FT	38
EP	28	UA	16	IS	3
matrix	0	heapsort	307	takfp	24
fannkuch	50	rijndaele	38	blowfishe	91
rijndaeld	12	blowfishd	105	sha	358
adpcmd	43	FFT	143	FFTi	213
adpcmc	340	patricia	347	dijkstra	148
susane	331	bitcount	47	susans	291
basicmath	69	qsort	149	susanc	258
lame	2	mad	3	jpegd	121
jpegc	205				

Tabela 5.4: Número de conjuntos com desempenho melhor que `-O3` para cada programa do espaço exploratório

Após este processo, o Seletor de Conjuntos extraiu apenas 20 conjuntos deste pequeno espaço exploratório e, finalmente, esse tamanho foi reduzido a apenas um conjunto, escolhido pelo Avaliador de Conjuntos.

Tais resultados evidenciam que a abordagem proposta é capaz de gerar um espaço de busca reduzido porém levanta a questão da qualidade das soluções presentes em tal espaço. Neste contexto, as próximas seções mostram que, de fato, a abordagem proposta aqui é capaz de alcançar ganhos de desempenho na maioria dos casos com este espaço exploratório.

5.5.2 Speedups

Para avaliação da qualidade das soluções geradas pelo *CBR-Selector*, foram realizados dois grupos de experimentos. Inicialmente foi utilizado um processo de validação *leave-one-out*, usando apenas programas pertencentes ao espaço exploratório (31 programas). Em seguida, foram utilizados 25 programas que não pertencem ao espaço exploratório. Portanto, nesse segundo experimento há conjuntos de treino e teste.

Para avaliação dos dois índices de similaridade propostos, os resultados foram separados em duas partes. A primeira apresenta os *speedups* alcançados com o uso do índice Jaccard Modificado e a segunda com o índice do Cosseno. Como cada índice foi experimentado com as duas estratégias de seleção, Probabilística e Justa, houve quatro

experimentos. Para cada experimento foram feitas as avaliações *leave-one-out* e treino e teste e, assim, serão apresentados oito resultados.

Jaccard

O uso do índice de Jaccard apontou melhores resultados na avaliação treino e teste (TT) do que na *leave-one-out* (LOO). O *speedup* médio alcançado foi de 1,085 para os programas TT e 1,065 para os programas LOO, o que indica um ganho de desempenho de 8,5% e 6,5%, respectivamente. Os *speedups* para cada programa são apresentados nas Figuras 5.8 e 5.9.

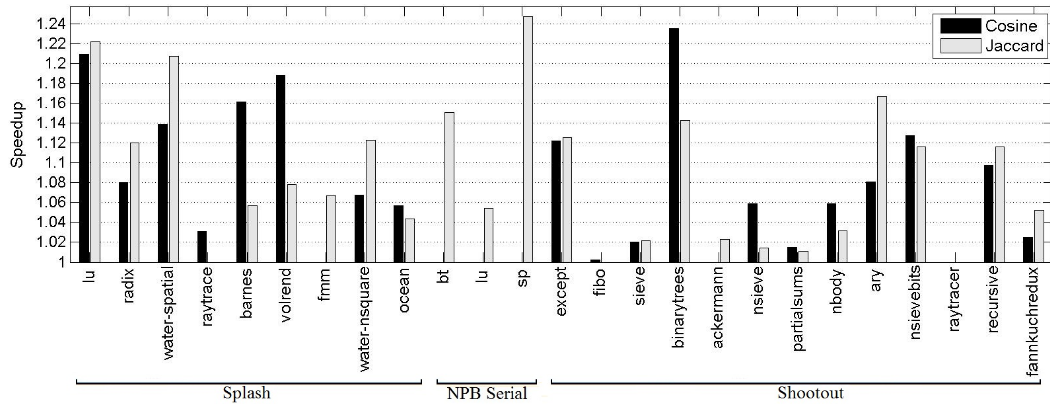
Este índice também apresentou impactos diferentes quando associado com as duas estratégias de seleção: probabilística e justa. O uso da abordagem `Jaccard/probabilística` no experimento LOO alcançou um *speedup* de 1,04 contra 1,09 do `Jaccard/justa`. Por outro lado, a mesma abordagem aplicada em TT apresentou um *speedup* de 1,09 contra 1,08 do `Jaccard/justa`. No entanto, apesar da `Jaccard/probabilística` apontar um *speedup* médio melhor para TT, o número de programas que alcançaram ganho de desempenho foi sempre maior para `Jaccard/justa` do que para `Jaccard/probabilística`. Em LOO, `Jaccard/probabilística` alcançou *speedup* para 16 dos 31 programas enquanto `Jaccard/justa` alcançou para 26 programas. Resultados semelhantes ocorreram em TT, onde `Jaccard/probabilística` alcançou *speedup* para 22 de 25 programas enquanto `Jaccard/justa` alcançou para 24 deles, sendo que apenas o `raytracer` do *Shootout* não alcançou *speedup*.

Além disso, o índice de Jaccard foi responsável pelo *speedup* máximo obtido, que ocorreu para `rijndaele` (1,75), em LOO com seleção justa. Ele também foi responsável pelo *speedup* máximo de TT com seleção justa e LOO com ambas seleções. Na seleção probabilística em LOO, `heapsort` (1,31) alcançou o *speedup* máximo. Em TT, `sp` alcançou o melhor *speedup* em ambas seleções *probabilística* (1,25) e justa (1,3).

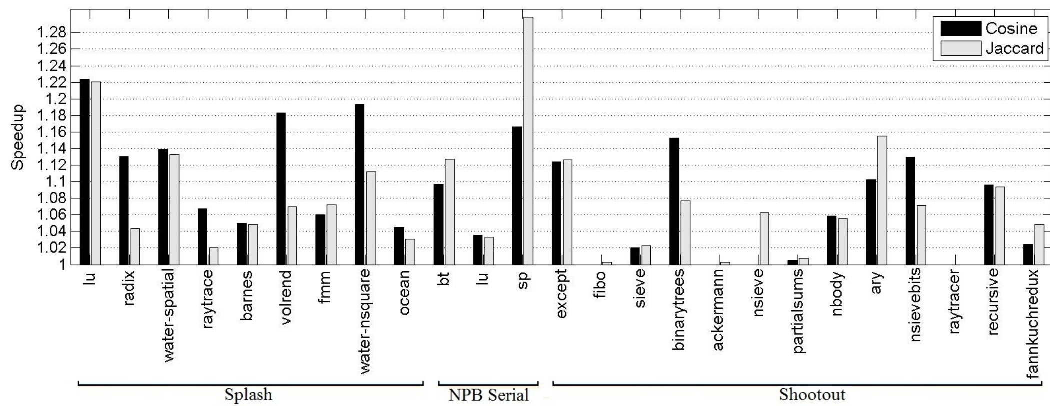
Ainda é interessante observar que em TT, o uso da seleção probabilística apenas alcançou *speedups* para `bt`, `lu`, `sp` e `fmm` quando associada ao índice Jaccard, ou seja, não obtiveram *speedup* na estratégia `Cosseno/probabilística`. O mesmo ocorreu para o `fmm` do *SPLASH-2*.

Cosseno

O índice de similaridade do Cosseno apresentou resultados similares ao índice de Jaccard em alguns pontos. Porém, houve algumas diferenças peculiares entre eles. Semelhante ao Jaccard, o Cosseno também apontou resultados melhores em TT do que em LOO. O



(a) Seleção Probabilística



(b) Seleção Justa

Figura 5.8: *Speedups* da avaliação Treino e Teste

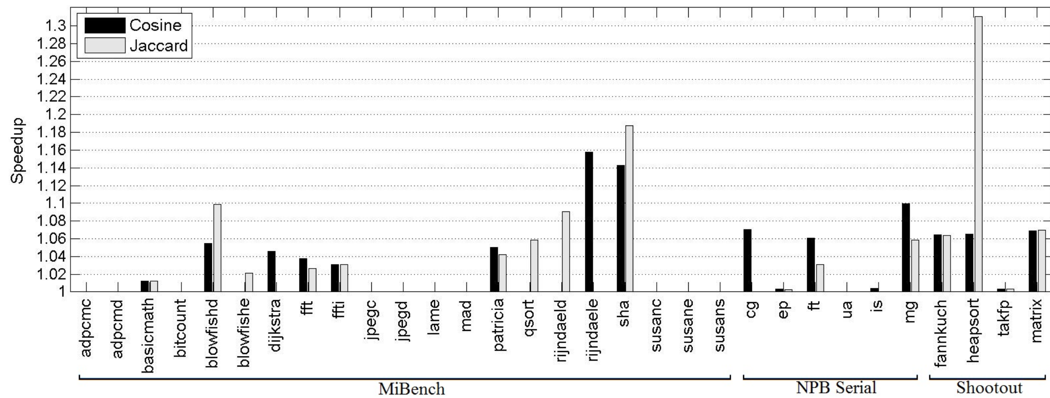
speedup médio alcançado com o uso do índice do Cosseno foi de 1,075 para TT e 1,07 para LOO. Os *speedups* para cada programa são apresentados nas Figuras 5.8 e 5.9.

Outro ponto similar ao índice Jaccard foi a diferença entre as seleções probabilística e justa. A `Cosseno/probabilística` apresentou um *speedup* médio de 1,03 e a `Cosseno/justa`, 1,1. Além disso, o número de programas que obtiveram ganho de desempenho foi sempre maior para a seleção justa do que para a probabilística. Em LOO, a `Cosseno/probabilística` alcançou *speedup* para 17 dos 31 programas enquanto a `Cosseno/justa` alcançou para 25. Para os programas TT, a `Cosseno/probabilística` alcançou *speedup* para 19 de 25 programas enquanto a `Cosseno/justa` alcançou para 21.

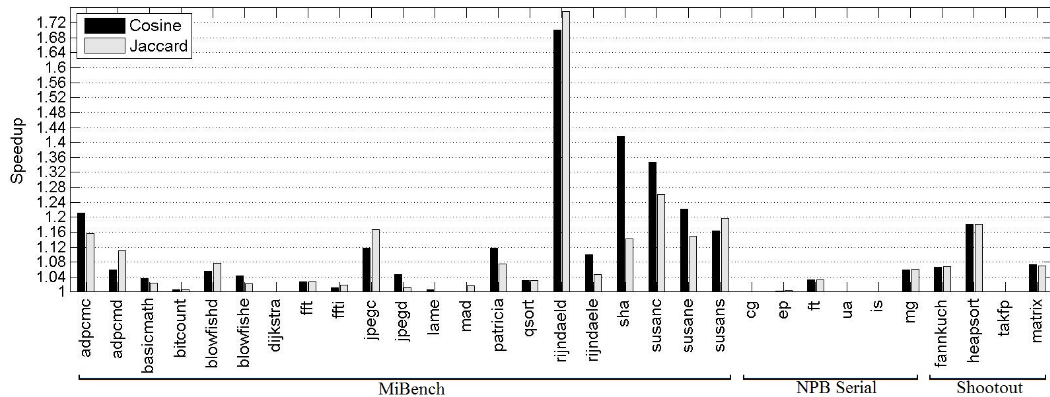
Apesar de o *speedup* médio alcançado pelo índice do Cosseno ter sido pior do que o alcançado pelo Jaccard para TT, ele foi melhor para LOO. O Cosseno também apresentou o maior *speedup* médio, que foi de 10% para seleção justa em LOO. O Cosseno também foi melhor para alguns casos particulares e, para alguns deles, apresentou diferenças significativas. No experimento TT com a seleção probabilística, isso aconteceu para 10 de

25 programas. Na seleção justa, isso ocorreu para 6 programas e o mesmo ocorreu para alguns casos em LOO.

Finalmente, houve programas que alcançaram *speedup* somente com uso do índice do Cosseno. No experimento TT com a seleção probabilística isso aconteceu para o `raytrace` do *SPLASH-2* e no experimento LOO isso ocorreu para os programas `dijkstra` e `cg`.



(a) Seleção Probabilística



(b) Seleção Justa

Figura 5.9: *Speedups* da avaliação *Leave-One-Out Cross Validation*

Discussão

Baseado nos resultados acima, há alguns pontos que devem ser destacados. O primeiro deles é o resultado geral, que sugere que o *CBR-Selector* é uma abordagem útil e eficiente. De fato, em um total de 56 programas, apenas o `ua` e o `raytracer` não obtiveram ganho algum de desempenho.

Outro ponto a ser observado é o resultado da seleção justa, que supera a probabilística tanto com o uso do índice de Jaccard quanto com o uso do índice do Cosseno. Esse

resultado indica que buscar menos conjuntos de mais variados programas é melhor do que buscar mais conjuntos de um grupo menor de programas, como faz a seleção probabilística.

Porém, os resultados também sugerem que a seleção probabilística é útil para alguns casos. Por exemplo, os programas `cg` e `dijkstra` somente alcançaram *speedup* com o uso da seleção probabilística.

Ainda é importante considerar que os resultados do índice de Jaccard parecem ser melhores que os do Cosseno, porém o uso do Cosseno foi melhor para um grande número de programas.

Estes pontos sugerem que o uso de diferentes estratégias é útil para encontrar bons resultados para o PSO, ainda que tais estratégias sejam semelhantes, com pequenas variações, como é o caso do *CBR-Selector*. Com frequência, conjuntos de otimizações que são alcançados por uma abordagem não são alcançados por outra e vice-versa. As variações realizadas exigiram apenas a implementação de duas estratégias de seleção diferentes e dois modelos de similaridade, sugerindo que a implementação de mais estratégias de seleção e modelos de similaridade deve explorar ainda mais o espaço exploratório construído, alcançando, consequentemente, *speedups* ainda mais altos.

5.5.3 Número de Avaliações

O maior diferencial da abordagem proposta nesta seção é o número de avaliações, que é constante para qualquer número de otimizações.

Enquanto as abordagens anteriores alcançaram *speedups* com um mínimo de 93, que foi o caso do IPBE, o *CBR-Selector*, para todos os casos, executou apenas 20 avaliações.

Se for considerado todo o processo, 500 avaliações foram necessárias para a construção do espaço exploratório e 20 avaliações para buscas nesse espaço, totalizando 520 avaliações. Ainda que o número de 520 avaliações seja menor que a maioria das abordagens anteriores, como a VNS e o PBE, é importante destacar o diferencial do *CBR-Selector*.

De fato, não é necessário executar um programa 520 vezes para encontrar bons *speedups* para ele. O processo que necessita de 500 avaliações pode ser considerado um processo "de fábrica", ou seja, ele pode ser executado antes de o compilador chegar às mãos do usuário final, que necessitará executar apenas 20 avaliações.

Tal abordagem é muito benéfica já que, considerando que toda a construção do espaço exploratório será feito "na fábrica", o tempo de sua construção não será crucial para o usuário final, podendo ser construído um espaço exploratório mais amplo com maior qualidade.

Em contraste com as demais abordagens, o melhor resultado das diferentes configurações do *CBR-Selector* foi um ganho médio de até 10,10% em um conjunto de 31 programas, ou seja, maior que o utilizado para as outras abordagens, e isso com apenas 20 avaliações.

Ainda, se for considerado apenas o conjunto *MiBench*, o *CBR-Selector* destaca-se com um ganho médio de 12,9% com a abordagem `Cosseno/probabilística`. Este resultado foi o que ficou mais próximo da VNS, que teve um ganho médio de 16,9%. No entanto, a diferença de número de avaliações entre uma abordagem e outra é grande, sugerindo que o uso de conhecimento prévio na solução do PSO é muito favorável.

5.6 Batch, Iterative and Combined Eliminations

As características peculiares do PSO motivaram a implementação de abordagens mais específicas para busca de soluções. Uma característica particular do PSO, por exemplo, é o fato de que nem todas otimizações favorecem o ganho de desempenho em todos programas. Conforme mencionado no Capítulo 2, uma otimização pode até mesmo prejudicar o desempenho de um determinado código para um dado objetivo. Portanto, pode ser explorada a busca por quais otimizações afetam negativamente o código para eliminá-las do conjunto.

Isto é o que procuram fazer as abordagens BE, IE e CE (BEIECE), apresentadas no Capítulo 3.

Considerando tais informações, esta seção apresenta a experimentação dos algoritmos BE, IE e CE aplicados ao PSO, que inclui a análise dos *speedups* para cada programa, bem como o número de avaliações utilizadas. O objetivo em tal análise é verificar se estes algoritmos são capazes de encontrar melhores soluções do que as encontradas pelas abordagens propostas.

5.6.1 Speedups

Neste experimento, todos os três algoritmos alcançaram ganhos significativos e para a maioria dos programas, chegando a um *speedup* de até 1,33 (CE/`susanc`). A Figura 5.10 apresenta o *speedup* para cada programa do *MiBench*.

Em média, o CE foi melhor do que os outros dois algoritmos, apresentando um ganho médio de 5,1% ao passo que o BE apresentou 2,7% e o IE 4,2%. Embora o CE tenha alcançado o melhor ganho de desempenho médio, os outros dois algoritmos superaram seu resultado para vários programas (`FFT`, `adpcmd`, `blowfishd`, `blowfishe`,

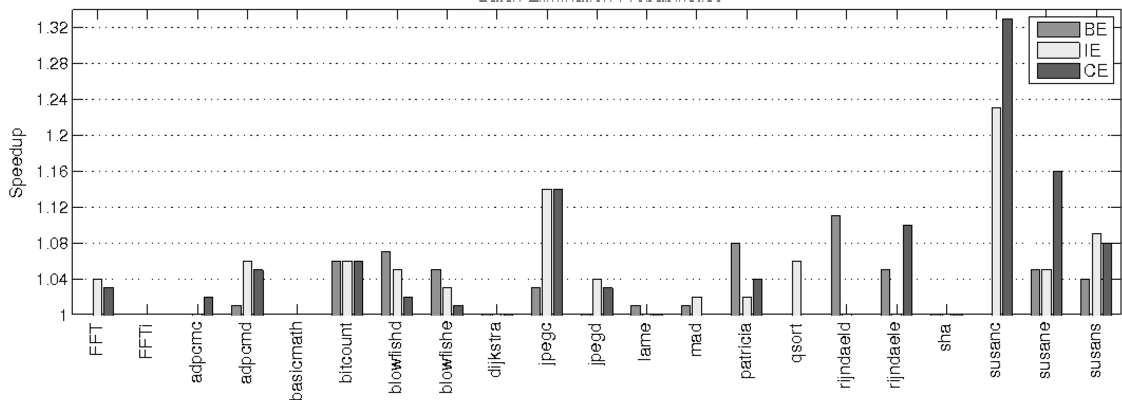


Figura 5.10: *Speedups* alcançados pelos algoritmos BE, IE e CE

`jpegd`, `mad`, `qsort`, `rijndaeld`, `susans`). O BE, apesar de ser um algoritmo mais rápido, trouxe os piores resultados. Ainda assim, ele superou as outras estratégias em alguns casos (`blowfishd`, `blowfishe`, `lame`, `patricia` e `rijndaeld`). Esses resultados indicam a importância de explorar estratégias diferentes, já que algumas, mesmo mais simples, podem encontrar soluções não alcançadas por estratégias mais elaboradas e complexas.

É importante mencionar também que os três algoritmos de Pan e Eigenmann (2006) são apresentados com as seguintes características: o BE como um algoritmo mais simples e inicial, a partir do qual os demais são construídos, o IE como uma exploração mais criteriosa e, consequentemente, mais morosa, e o CE como um algoritmo mais sofisticado que contempla uma combinação dos outros dois algoritmos. Um ponto a ser destacado aqui é que os *speedups* alcançados refletem diretamente estas características, já que em média o IE foi melhor que o BE e o CE, por sua vez, foi melhor que o IE.

Ainda devem ser observadas as semelhanças e diferenças entre os resultados do BEIECE (Figura 5.10) e os das outras abordagens. Por exemplo, alguns dos programas que obtiveram os melhores *speedups* foram os mesmos tanto na VNS quanto no BEIECE, como é o caso do `susanc` e `susane`. O fato de conjuntos muito bons terem sido encontrados em ambas abordagens reforça o argumento de que alguns programas oferecem melhores oportunidades de otimização que outros, a tal ponto que, em abordagens bem diferentes, se destacam de modo semelhante. No caso da VNS e do BEIECE, o mesmo pode ser observado nos piores casos, que podem ser exemplificados com o `basicmath` e o `lame`, que não alcançaram (ou alcançaram muito baixos) *speedups* em ambas abordagens.

Por fim, em essência, é possível observar que, mesmo alcançando *speedups* significativos, o melhor *speedup* médio dos três algoritmos apresentados aqui (CE) foi menor do que a maioria das abordagens propostas e avaliadas neste trabalho.

5.6.2 Número de Avaliações

A Figura 5.11 apresenta o número de avaliações executadas até o final de cada algoritmo para cada programa e a Figura 5.12 apresenta a relação entre o ganho de desempenho e o número de avaliações. O BE necessita de uma quantidade de avaliações igual ao tamanho do conjunto padrão. O conjunto padrão selecionado aqui apresenta tamanho 63. Já no IE e no CE o número de avaliações dependerá das decisões que o algoritmo tomar, podendo chegar a 63^2 avaliações. Conforme esperado, na maioria das vezes o CE executou mais rapidamente que o IE, e ainda trazendo melhores resultados.

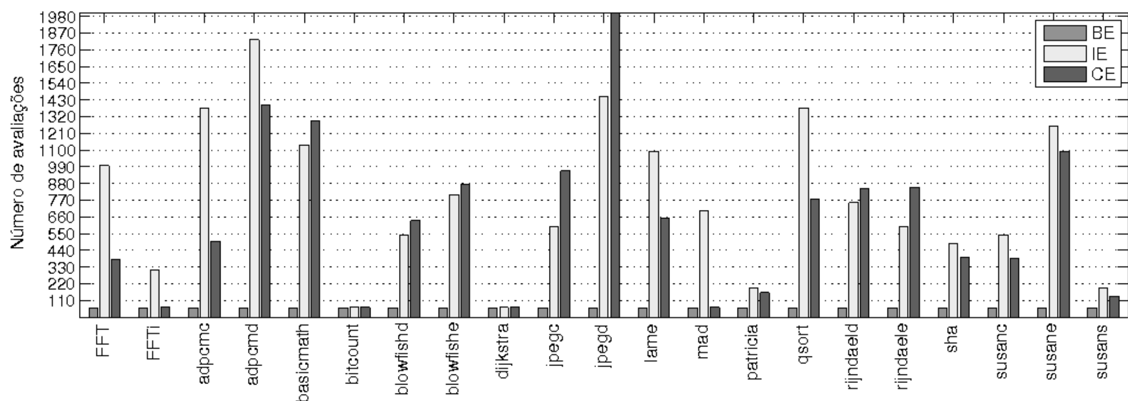


Figura 5.11: Número de avaliações para as soluções encontradas pelo BE, IE e CE

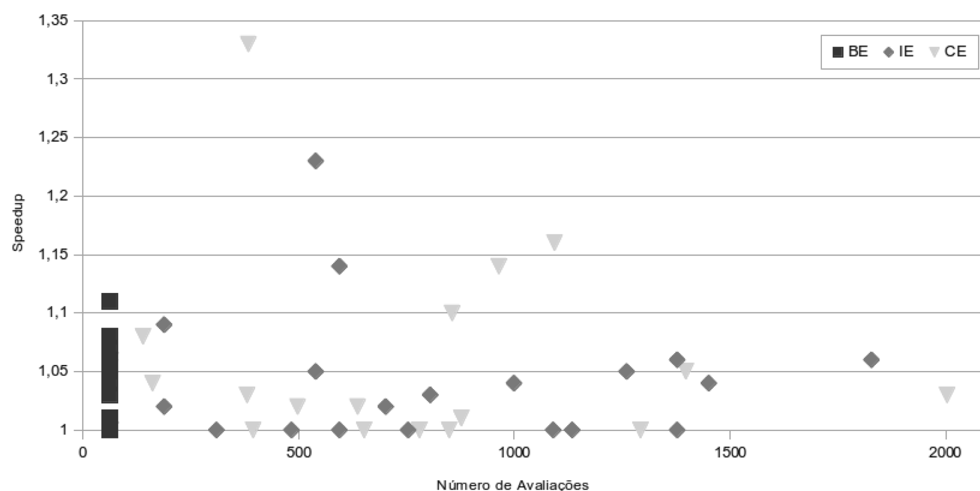


Figura 5.12: *Speedups* e número de avaliações do BE, IE e CE

Os resultados indicam que estes algoritmos podem tanto encontrar uma solução rapidamente, quanto necessitar de um longo tempo de execução para encontrar uma solução. Os casos em que o CE encontrou uma solução rapidamente (65 avaliações), esta

solução não apresentou ganhos de desempenho significativos, como para os programas `FFTi`, `bitcount`, `dijkstra` e `mad`. No entanto, o CE apresentou um ganho de desempenho relativamente alto para `susanc`, `susane`, e `jpegc`, com 384, 1093 e 964 avaliações, respectivamente. O mesmo ocorre com o IE, que teve um ganho alto, porém menor que o CE, para `jpegc` e `susanc`, com 595 e 540 avaliações, respectivamente. Isso evidencia que esses algoritmos, apesar de executarem rapidamente em alguns casos, apenas geram soluções de qualidade após um número significativo de avaliações.

Tais resultados sugerem que, neste contexto, é melhor utilizar uma abordagem probabilística, como o PBE por exemplo, ao invés de uma abordagem agressiva, como o algoritmo BE original, que elimina todas otimizações que isoladamente prejudicam o desempenho do programa em uma única etapa. Também há vantagens em utilizar uma abordagem probabilística ao invés de abordagens determinísticas como IE e CE, já que os resultados do PBE foram mais favoráveis que qualquer um destes três algoritmos.

5.7 Resumo das Abordagens Avaliadas

A Tabela 5.5 apresenta um resumo dos resultados de cada uma das abordagens avaliadas. Para cada abordagem são apresentados: a complexidade, o número médio de avaliações, o ganho de desempenho médio e o número de programas para os quais a abordagem alcançou *speedup*. A Figura 5.13 mostra a relação entre o ganho de desempenho médio e o número de avaliações médio para cada abordagem.

Os dados da Tabela 5.5 e da Figura 5.13 permitem uma visão geral de todas as abordagens, de modo que algumas comparações podem ser feitas entre elas.

Considerando inicialmente a VNS, esta alcançou o maior ganho de desempenho, com uma média de 16,9% e atingindo 100% dos programas avaliados. No entanto, como mencionado anteriormente, ao custo de um alto tempo de resposta. Isso não exclui sua aplicação, já que, apesar de morosa, deve encontrar *speedups* para programas que as outras abordagens não encontram.

As abordagens mais específicas BE, IE e CE mostraram ser mais rápidas que a VNS, porém suas médias de ganho de desempenho de 2,71%, 4,24% e 5,1%, respectivamente, foram mais baixas do que aquelas obtidas pela VNS. Além disso, o número de programas que essas abordagens atingiu foi pequeno, sendo pouco mais da metade.

O algoritmo PBE apresentou um grande diferencial, superando o BEIECE e pareceu apresentar uma boa alternativa para seleção de otimizações, com uma melhoria de 9,33% com um número de avaliações (630) menor do que do CE (649) e do IE (778). Além disso, enquanto o BEIECE atingiu pouco mais da metade dos programas, o PBE

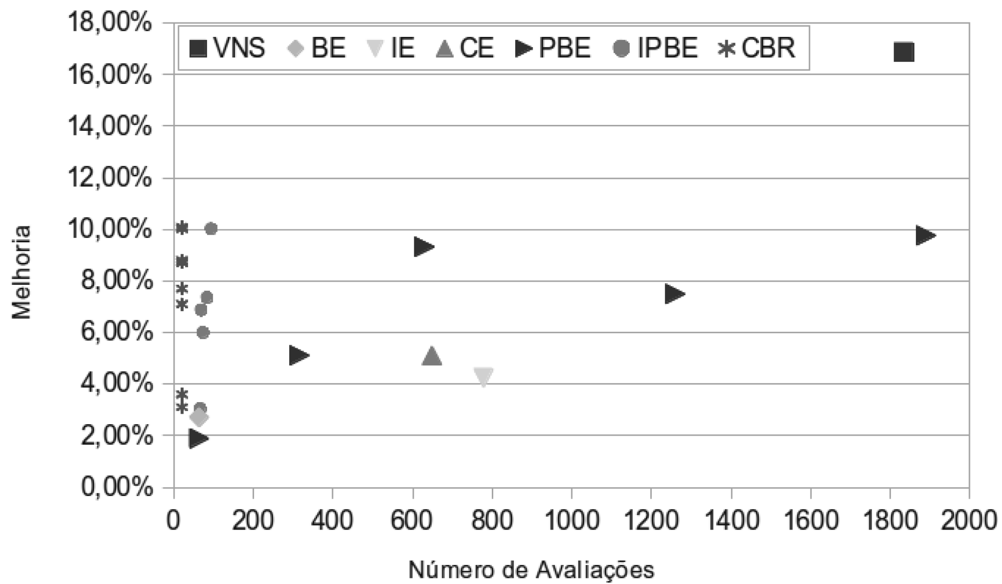


Figura 5.13: Comparação ganho de desempenho *versus* número de avaliações para todas abordagens experimentadas

alcançou *speedup* para 20 de 21 programas, ou seja, quase a mesma amplitude da VNS. É interessante observar o comportamento do PBE, que geralmente melhora seu resultado conforme o número de avaliações aumenta. No entanto, os resultados sugeriram que o ganho de desempenho não cresce na mesma proporção do aumento das avaliações. Com 1890 avaliações (que equivale a 30 repetições do PBE), o ganho foi de 9,76%. Um ganho bem maior foi alcançado com a VNS com um número de avaliações menor que esse (1835). No entanto, é importante mencionar que o número de avaliações do PBE será sempre o mesmo para qualquer programa, enquanto o número de avaliações da VNS varia de acordo com o programa. Assim, pode haver casos em que será melhor utilizar o PBE do que a VNS.

Em termos de *speedup* médio, o IPBE supera todas as estratégias citadas anteriormente, com exceção da VNS. Os resultados obtidos por esse algoritmo sugerem que ele, apesar de atingir menos programas que a VNS e o PBE, apresenta grande viabilidade para ser utilizado no PSO, já que alcançou um ganho médio de 10,02%, atingindo 19 programas. O maior destaque do IPBE, no entanto, é que ele alcançou tais resultados com apenas 93 avaliações, um número muito menor que qualquer uma das abordagens citadas acima.

Já o *CBR-Selector* apresentou uma viabilidade ainda maior, alcançando um ganho de desempenho médio de até 10,10% e atingindo 25 de 31 programas. As variações de estratégias de seleção e índice de similaridade do seletor mostraram diferentes resultados,

Abordagem	Complex.	Avaliações	Ganho	No. Speedups
VNS	$\Omega(NB)$ [a]	1835	16,87%	21 de 21
PBE - 1	$\Theta(cn)$	63	1,89%	12 de 21
PBE - 5	$\Theta(cn)$	315	5,11%	18 de 21
PBE - 10	$\Theta(cn)$	630	9,33%	19 de 21
PBE - 20	$\Theta(cn)$	1260	7,49%	19 de 21
PBE - 30	$\Theta(cn)$	1890	9,76%	20 de 21
IPBE - 3	$\Theta(c+n)$	66	3,04%	10 de 21
IPBE - 5	$\Theta(c+n)$	68	6,87%	18 de 21
IPBE - 10	$\Theta(c+n)$	73	6,00%	18 de 21
IPBE - 20	$\Theta(c+n)$	83	7,36%	17 de 21
IPBE - 30	$\Theta(c+n)$	93	10,02%	19 de 21
CBR/JP LOO	$\Theta(L)$	20	3,60%	16 de 31
CBR/JJ LOO	$\Theta(L)$	20	8,70%	26 de 31
CBR/CP LOO	$\Theta(L)$	20	3,10%	17 de 31
CBR/CJ LOO	$\Theta(L)$	20	10,10%	25 de 31
CBR/JP TT	$\Theta(L)$	20	8,80%	22 de 25
CBR/JJ TT	$\Theta(L)$	20	7,70%	24 de 25
CBR/CP TT	$\Theta(L)$	20	7,10%	19 de 25
CBR/CJ TT	$\Theta(L)$	20	8,40%	21 de 25
BE	$\Theta(n)$	63	2,71%	12 de 21
IE	$O(n^2)$	778	4,24%	13 de 21
CE	$O(n^2)$	649	5,10%	13 de 21

[a] Onde N = número de iterações e B = custo da busca local

Tabela 5.5: Quadro comparativo dos resultados das abordagens avaliadas

sendo que alguns atingem maior número de programas e outros apresentam um *speedup* médio maior. Porém, em todas estratégias o número de avaliações foi de apenas 20. Esta característica faz do *CBR-Selector* uma estratégia viável para a maioria das instâncias do PSO, já que executa muito mais rapidamente que todas as outras abordagens e alcança bons *speedups*.

Outro ponto que não pode ser desconsiderado é que a experimentação do *CBR-Selector* foi mais exaustiva que das outras abordagens, que utilizaram apenas o *MiBench* na experimentação. Se for considerado apenas o conjunto *MiBench*, o *CBR-Selector* alcançou 12,9% de ganho de desempenho e atingiu 19 dos 21 programas na avaliação *Leave-One-Out* com a seleção justa e índice de similaridade cosseno, superando todas as demais estratégias, com exceção da VNS. Porém, a eficiência do *CBR-Selector* o coloca à frente da VNS, que necessitou de, em média, 1835 avaliações contra apenas 20 do *CBR-Selector*.

5.8 Considerações Gerais

Este capítulo apresentou a avaliação experimental de cada abordagem desenvolvida no presente trabalho, comparando-as com a abordagem proposta por Pan e Eigenmann (2006). As abordagens desenvolvidas superaram o desempenho dos algoritmos BE, IE e CE em termos de ganho de desempenho, apesar de algumas necessitarem um número um maior de avaliações. Essa experimentação, além de avaliar o número de avaliações e o *speedup* para cada abordagem, sugere a aplicabilidade de cada uma delas: a VNS pode ser aplicada em contextos em que a prioridade é a qualidade do código gerado independente do tempo de resposta, enquanto o *CBR-Selector* pode ser utilizado em contextos em que a resposta deve ser rápida, por exemplo. No entanto, o objetivo principal da experimentação realizada neste capítulo foi validar as abordagens desenvolvidas e fornecer subsídios para as conclusões apresentadas no capítulo seguinte.

6

Conclusões e Trabalhos Futuros

O presente trabalho teve como foco o Problema da Seleção de Otimizações (PSO) citado na literatura. Dada a necessidade de produzir códigos com mais qualidade, os projetistas de compiladores implementarem dezenas de otimizações. No entanto, a complexidade das transformações efetuadas no código pelas otimizações não favorece um entendimento pleno da interação entre elas. Assim, para garantir o melhor conjunto de otimizações para determinado código não se conhece outra solução a não ser avaliar cada possibilidade.

Porém, foi apresentado no Capítulo 2 que a exploração de todas as possibilidades de aplicação de otimizações é inviável na maioria das aplicações e portanto há uma motivação para a investigação de abordagens alternativas a essa. O Capítulo 3 apresentou trabalhos na literatura que utilizaram abordagens que envolveram o uso de buscas exaustivas e aleatórias, técnicas estatísticas, eliminação iterativa, algoritmos genéticos e aprendizagem de máquina na busca de soluções viáveis para o problema.

O objetivo desses trabalhos em geral foi determinar uma meta específica e procurar um conjunto de otimizações que proporcionava um ganho de desempenho, em relação a um conjunto padrão utilizado pelos compiladores, considerando tal meta. Esse objetivo foi alcançado em todos os trabalhos, em maior ou menor qualidade.

Tal análise dos trabalhos das literatura forneceu subsídios para que fossem propostas quatro novas abordagens para o problema, as quais foram apresentadas no Capítulo 4. Esses trabalhos apresentam semelhanças e diferenças em relação às abordagens aqui propostas, sendo mencionadas ao final do referido capítulo.

As abordagens propostas para o PSO foram as seguintes: um algoritmo de busca no espaço exploratório do PSO com a metaheurística VNS considerando operadores de

busca local adequados para o problema, o algoritmo probabilístico PBE e o algoritmo IPBE, ambos baseados no *Batch Elimination* de Pan e Eigenmann (2006) e, por fim, o *CBR-Selector* com uso de raciocínio baseado em casos, com quatro configurações de índices de similaridade e estratégias de seleção.

Para mostrar a validade das abordagens implementadas foi realizada uma avaliação experimental nelas e também nos algoritmos *Batch Elimination*, *Iterative Elimination* e *Combined Elimination*, de Pan e Eigenmann (2006), para fins comparativos. O maior *speedup* médio foi alcançados pela VNS que, no entanto, foi muito lenta. Os algoritmos PBE e IPBE apresentaram resultados promissores, já que alcançaram *speedups* maiores que todos os algoritmos originais (BE, IE e CE) e com um número inferior de avaliações. Por fim, o *CBR-Selector* se destacou por ter alcançado o maior *speedup* por número de avaliações. O uso de diferentes índices de similaridade e diferentes estratégias de seleção no *CBR-Selector* foi essencial para que esse resultado fosse alcançado.

A avaliação experimental realizada também confirma as contribuições deste trabalho. Primeiramente, a experimentação permite concluir que as quatro abordagens desenvolvidas são viáveis em contextos diversos. Além disso, os resultados evidenciaram que os operadores de busca local definidos para VNS foram adequados ao PSO, já que *speedups* altos foram alcançados com essa abordagem. As modificações nos algoritmos BE, IE e CE também apresentaram vantagens e são relevantes. Por fim, os resultados do *CBR-Selector* mostraram a possibilidade de selecionar otimizações com um número relativamente baixo de iterações, alcançando ganhos de desempenho satisfatórios.

6.1 Trabalhos Futuros

Com o objetivo de fornecer contribuições ao PSO, alguns trabalhos futuros são propostos:

Maior experimentação da VNS, BEIECE, PBE e IPBE A experimentação dessas estratégias utilizou o *MiBench*, que fornece aplicações utilizadas no cotidiano. Para ampliar a validade dessas abordagens os mesmo experimentos devem ser executados com outras categorias de *benchmarks*, como os científicos;

Otimização da VNS Configurar a VNS para efetuar um número menor de buscas locais e avaliar a qualidade dos resultados. Como a principal desvantagem da VNS é a sua morosidade, é razoável avaliar essa possibilidade;

Solucionar o PSO com outras metaheurísticas Como a VNS apresentou *speedups* muito altos é relevante avaliar outras metaheurísticas aplicadas ao PSO;

Distribuição diferente de pontos no IPBE O algoritmo IPBE distribui os pontos em intervalos equidistantes de acordo com o número de tentativas. Podem ser exploradas outras formas de realizar essa distribuição como uma distribuição probabilística tendenciosa a um ponto específico;

Maior análise de pontos do IPBE Como pôde ser percebido nos resultados do IPBE, alguns intervalos apresentam um maior agrupamento de *speedups*. É também relevante avaliar em um número maior de *benchmarks* a possibilidade de existir um agrupamento maior próximo de um determinado ponto. Se alguma peculiaridade de agrupamento for detectada será conveniente utilizar um valor de probabilidade fixo, reduzindo ainda mais o tempo de execução do IPBE;

Metodologia de geração do espaço exploratório do *CBR-Selector* A metodologia utilizada para a construção do espaço exploratório foi simples, tendo sido feita com conjuntos gerados aleatoriamente. Algumas estratégias podem melhorar a construção desse espaço. Há sugestões na literatura para ordens de otimizações (Muchnick, 1997). Na construção do espaço exploratório, essas sugestões podem ser consideradas como critérios para não permitir a avaliação com conjuntos que não atendam a esses critérios. Além dessa estratégia, podem ser exploradas outras técnicas de busca como o uso de algoritmos genéticos ou metaheurísticas;

Maior exploração das informações de *performance counters* As informações de *performance counters* podem ser melhor exploradas no contexto de seleção de otimizações. Uma proposta é coletá-los para cada uma das otimizações e procurar qual o subconjunto de otimizações que minimiza os *performance counters*.

Metodologia para melhoria da ordem de aplicação Como a questão da ordem da aplicação de otimizações não foi abordada diretamente neste trabalho, ela deve ser considerada em trabalhos futuros. Uma proposta inicial é melhorar os conjuntos encontrados pelas abordagens apresentadas, explorando as possibilidades de ordem de aplicação das otimizações.

6.2 Considerações Finais

Todas abordagens propostas e implementadas no presente trabalho alcançaram *speedups* significativos e superaram, em pelo menos uma configuração, a abordagem da literatura implementada para avaliação (Pan e Eigenmann, 2006). Além disso, a forma como essas abordagens foram apresentadas e avaliadas possibilita uma análise ampla sobre

o Problema da Seleção de Otimizações, bem como o entendimento das semelhanças e diferenças entre elas e suas respectivas vantagens e desvantagens.

O seletor que utilizou Raciocínio Baseado em Casos aqui proposto, denominado *CBR-Selector*, apresentou-se como vantajoso diante de trabalhos recentes na literatura, pois alcançou *speedups* em um conjunto de 56 *benchmarks* com uma implementação simples e objetiva.

A experimentação do *CBR-Selector* também reforçou ainda mais a ideia de que o uso de aprendizagem de máquina é necessário quando o objetivo é prever bons conjuntos de otimização com um número baixo de avaliações.

Por fim, a natureza diversa das abordagens propostas mostra quão amplo é o problema abordado que pode e deve ser explorado com o maior número de estratégias possível para que melhores soluções sejam encontradas.

Referências

AAMODT, A.; PLAZA, E. Case-based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, v. 7, n. 1, p. 39–59, 1994.

ADHIANTO, L.; BANERJEE, S.; FAGAN, M.; KRENTEL, M.; MARIN, G.; MELLOR-CRUMMEY, J.; TALLENT, N. R. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computing : Practice and Experience*, v. 22, n. 6, p. 685–701, 2010.

AGRAWAL, R.; SRIKANT, R. Fast Algorithms for Mining Association Rules in Large Databases. In: *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, p. 487–499.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. 2 ed. Boston, MA, USA: Prentice Hall, 2006.

ALMAGOR, L.; COOPER, K. D.; GROSUL, A.; HARVEY, T. J.; REEVES, S. W.; SUBRAMANIAN, D.; TORCZON, L.; WATERMAN, T. Finding Effective Compilation Sequences. *SIGPLAN Notices*, v. 39, n. 7, p. 231–239, 2004.

BAILEY, D. H.; BARSZCZ, E.; BARTON, J. T.; BROWNING, D. S.; CARTER, R. L.; DAGUM, L.; FATOOHI, R. A.; FREDERICKSON, P. O.; LASINSKI, T. A.; SCHREIBER, R. S.; SIMON, H. D.; VENKATAKRISHNAN, V.; WEERATUNGA, S. K. The nas parallel benchmarks – summary and preliminary results. In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, New York, NY, USA: ACM, 1991, p. 158–165.

BEASLEY, D.; BULL, D. R.; MARTIN, R. R. An Overview of Genetic Algorithms: Part 1, Fundamentals. *University Computing*, v. 15, p. 58–69, 1993.

BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHANG, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.;

Stefanović, D.; VanDrunen, T.; von Dincklage, D.; Wiedermann, B. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Notices*, v. 41, n. 10, p. 169–190, 2006.

Blum, C.; Roli, A. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, v. 35, n. 3, p. 268–308, 2003.

Blum, R. *Professional Assembly Language*. Programmer to Programmer. Indianapolis, Indiana, USA: Wiley, 2005.

Bungo, J. The Use of Compiler Optimizations for Embedded Systems Software. *Crossroads*, v. 15, p. 8–15, 2008.

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 International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2007, p. 185–197.

Cavazos, J.; Moss, J. E. B.; O'Boyle, M. F. P. Hybrid Optimizations: Which Optimization Algorithm to Use? In: *Compiler Construction*, Berlin, Heidelberg: Springer, 2006, p. 124–138.

Cavazos, J.; O'Boyle, M. F. P. Method-specific Dynamic Compilation Using Logistic Regression. *SIGPLAN Notices*, v. 41, n. 10, p. 229–240, 2006.

Chang, C.-C.; Lin, C.-J. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, v. 2, n. 3, p. 27:1–27:27, 2011.

Collins, W. R. Tasking solutions to the sieve of eratosthenes. *Ada Letters*, v. XVIII, n. 4, p. 107–110, 1998.

Cooper, K. D.; Schielke, P. J.; Subramanian, D. Optimizing for Reduced Code Space Using Genetic Algorithms. *SIGPLAN Notices*, v. 34, n. 7, p. 1–9, 1999.

Dongarra, J.; London, K.; Moore, S.; Mucci, P.; Terpstra, D. Using PAPI for Hardware Performance Monitoring on Linux Systems. In: *Proccedings of the Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, 2001.

Dongarra, J. J.; Luszczek, P.; Petitet, A. The LINPACK Benchmark: Past, Present and Future. *Concurrency and Computation: Practice and Experience*, v. 15, n. 9, p. 803–820, 2003.

DRAPER, N. R.; SMITH, H. *Applied Regression Analysis.* Wiley Series in Probability and Statistics, 3 ed. Wiley-Interscience, 1998.

FOLEISS, J. H.; DA SILVA, A. F.; RUIZ, L. B. An Experimental Evaluation of Compiler Optimizations on Code Size. In: *Proceedings of the Brazilian Symposium on Programming Languages*, São Paulo, São Paulo, Brazil: EACH USP, 2011a, p. 1–15.

FOLEISS, J. H.; DA SILVA, A. F.; RUIZ, L. B. The Effect of Combining Compiler Optimizations on Code Size. In: *Proceedings of the International Conference of the Chilean Computer Science Society*, Curicó, Chile: Sociedad Chilena de Ciencias de la Computación, 2011b, p. 1–8.

FORSYTHE, G. E.; MALCOLM, M. A.; MOLER, C. B. *Computer methods for mathematical computations.* Prentice-Hall series in automatic computation. Englewood Cliffs N.J: Prentice-Hall, 1977.

FRYZA, T. Basic C Code Implementations for AVR Microcontrollers. In: *Systems, Signals and Image Processing, 2007 and 6th EURASIP Conference focused on Speech and Image Processing, Multimedia Communications and Services. 14th International Workshop on*, Maribor, Slovenia: IEEE, 2007, p. 434 –437.

GENDREAU, M.; POTVIN, J.-Y. *Handbook of metaheuristics*, cáp. Variable Neighborhood Search. 2nd ed Springer Publishing Company, Incorporated, 2010.

GERÔNIMO, J. R.; FRANCO, V. S. *Fundamentos da Matemática: Uma Introdução à Lógica Matemática, Teoria dos Conjuntos, Relações e Funções.* EDUEM, 2008.

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 International Workshop of Workload Characterization*, Washington, DC, USA: IEEE Computer Society, 2001, p. 3–14.

HANEDA, M.; KNIJNENBURG, P. M. W.; WIJSHOFF, H. A. G. Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics. In: *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA: IEEE Computer Society, 2005, p. 123–132.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fourth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

HOLLANDER, M.; WOLFE, D. A. *Nonparametric Statistical Methods.* 2 ed. California, USA: Wiley-Interscience, 1999.

JAVADIAN, N.; MOZDGIR, A.; KOUHI, E.; QAJAR, D.; SHIRAQAI, M. Solving Assembly Flowshop Scheduling Problem with Parallel Machines Using Variable Neighborhood Search. In: *Computers Industrial Engineering, 2009. CIE 2009. International Conference on*, 2009, p. 102–107.

KUFRIN, R. Measuring and Improving Application Performance with PerfSuite. *Linux J.*, v. 2005, n. 135, p. 4–, 2005.

KULKARNI, P. A.; HINES, S. R.; WHALLEY, D. B.; HISER, J. D.; DAVIDSON, J. W.; JONES, D. L. Fast and Efficient Searches for Effective Optimization-Phase Sequences. *ACM Transactions on Architecture and Code Optimization*, v. 2, n. 2, p. 165–198, 2005.

KULKARNI, P. A.; WHALLEY, D. B.; TYSON, G. S.; DAVIDSON, J. W. Exhaustive Optimization Phase Order Space Exploration. In: *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2006, p. 306–318.

KULKARNI, S.; CAVAZOS, J. Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning. *SIGPLAN Notices*, v. 47, n. 10, p. 147–162, 2012.

KUMAR, R. V.; NARAYANAN, B. L.; GOVINDARAJAN, R. Dynamic Path Profile Aided Recompilation in a JAVA Just-In-Time Compiler. In: *Proceedings of the 9th International Conference on High Performance Computing*, London, UK, UK: Springer-Verlag, 2002, p. 495–505.

LATTNER, C. *LLVM: An Infrastructure for Multi-Stage Optimization.* Dissertação de Mestrado, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, 2002.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California, 2004.

LAU, J.; ARNOLD, M.; HIND, M.; CALDER, B. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, 2006, p. 239–251.

LEATHER, H.; BONILLA, E.; O'BOYLE, M. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In: *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2009, p. 81–91.

LEE, C.; POTKONJAK, M.; MANGIONE-SMITH, W. H. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, Washington, DC, USA: IEEE Computer Society, 1997, p. 330–335.

LEI-FU, G.; WEI, D. A Parallel Variable Neighborhood Search for the Traveling Salesman Problem. In: *Advanced Management Science (ICAMS), 2010 IEEE International Conference on*, 2010, p. 150–152.

LINDHOLM, T.; YELLIN, F. *Java Virtual Machine Specification.* 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

MARKOV, Z.; RUSSELL, I. An introduction to the weka data mining system. *SIGCSE Bulletin*, v. 38, n. 3, p. 367–368, 2006.

MILLER, F. P.; VANDOME, A. F.; MCBREWSTER, J. *Inline Expansion: Compiler Optimization.* Worldwide: Alpha Press, 2010.

MLADENOVIĆ, N. A Variable Neighborhood Algorithm – A New Metaheuristics for Combinatorial Optimization. In: *Abstracts of Papers Presented at Optimization Days. Montreal*, 1995.

MUCHNICK, S. S. *Advanced Compiler Design and Implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

NGUYEN, T. V. N.; IRIGOIN, F. Efficient and Effective Array Bound Checking. *ACM Transactions on Programming Languages and Systems*, v. 27, n. 3, p. 527–570, 2005.

PAN, Z.; EIGENMANN, R. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In: *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2006, p. 319–332.

PARK, E.; KULKARNI, S.; CAVAZOS, J. An Evaluation of Different Modeling Techniques for Iterative Compilation. In: *Proceedings of the International Conference*

on Compilers, Architectures and Synthesis for Embedded Systems, New York, NY, USA: ACM, 2011, p. 65–74.

PEREZ, G. A.; KAO, C.-M.; CHUNG, Y.-C.; HSU, W.-C. A Hybrid Just-in-Time Compiler for Android: Comparing JIT Types and the Result of Cooperation. In: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, New York, NY, USA: ACM, 2012, p. 41–50.

PURINI, S.; JAIN, L. Finding Good Optimization Sequences Covering Program Space. *ACM Transactions on Architecture and Code Optimization*, v. 9, n. 4, p. 56:1–56:23, 2013.

RUSSELL, S. J.; NORVIG, P.; CANDY, J. F.; MALIK, J. M.; EDWARDS, D. D. *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

SAVINIEC, L.; CONSTANTINO, A. *Operadores de Vizinhança Eficientes para Algoritmos de Busca Local Aplicados ao Problema de Horários em Escolas.* Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação - Universidade Estadual de Maringá, Maringá, 2013.

SCOTT, M. L. *Programming Language Pragmatics.* 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

SEBESTA, R. W. *Concepts of Programming Languages.* 9th ed. USA: Addison-Wesley Publishing Company, 2009.

SMITH, J.; NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

STALLMAN, R. M.; DEVELOPERCOMMUNITY, G. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3.* Paramount, CA: CreateSpace, 2009.

STANLEY, K. O.; MIIKKULAINEN, R. Efficient Reinforcement Learning Through Evolving Neural Network Topologies. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, San Francisco: Morgan Kaufmann, 2002, p. 9.

STEPHENSON, M.; AMARASINGHE, S.; MARTIN, M.; O'REILLY, U.-M. Meta Optimization: Improving Compiler Heuristics with Machine Learning. *SIGPLAN Notices*, v. 38, n. 5, p. 77–90, 2003.

Tan, P.-N.; Steinbach, M.; Kumar, V. *Introduction to Data Mining.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

Triantafyllis, S.; Vachharajani, M.; Vachharajani, N.; August, D. I. Compiler Optimization-Space Exploration. In: *Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2003, p. 204–215.

Woo, S. C.; Ohara, M.; Torrie, E.; Singh, J. P.; Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Computer Architecture News*, v. 23, n. 2, p. 24–36, 1995.

Würthinger, T.; Wimmer, C.; Mössenböck, H. Array Bounds Check Elimination for the Java HotSpot Client Compiler. In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, New York, NY, USA: ACM, 2007, p. 125–133.

Zhang, G. Hybrid Variable Neighborhood Search for Multi Objective Flexible Job Shop Scheduling Problem. In: *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, 2012, p. 725–729.