

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

JOÃO FABRÍCIO FILHO

**Estratégias para Exploração de Sequências de Transformações
do Compilador**

Maringá
2017

JOÃO FABRÍCIO FILHO

**Estratégias para Exploração de Sequências de Transformações
do Compilador**

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
2017

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

F126e	<p>Fabrcio Filho, Joao</p> <p>Estratégias para exploração de sequências de transformações do compilador/ . -- Maringá, 2017. 89 f. : il. color, figs., tabs.</p> <p>Orientador: Prof. Dr. Anderson Faustino da Silva.</p> <p>Dissertação (mestrado) - Universidade Estadual de Maringá, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação, 2017.</p> <p>1. Sistemas de computação. 2. Compiladores. 3. Caracterização de Programas. 4. Raciocínio baseado em casos. 5. Compilação interativa. I. Silva, Anderson Faustino da, orient. II. Universidade Estadual de Maringá. Centro de Tecnologia. Programa de Pós-Graduação em Ciência da computação. III. Título.</p> <p>CDD 22. ED.005.453 JLM001622</p>
-------	---


FOLHA DE APROVAÇÃO

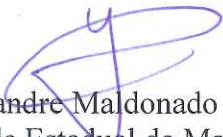
JOÃO FABRÍCIO FILHO


Estratégias para exploração de sequências de transformações do compilador

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


Prof. Dr. Yandre Maldonado e Gomes da Costa
Universidade Estadual de Maringá – DIN/UEM


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

Aprovada em: 17 de fevereiro de 2017.

Local da defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá.

AGRADECIMENTOS

Agradeço aos meus pais, João (*in memoriam*) e Helena, pela educação e pelo apoio e incentivo aos estudos que sempre tive.

Agradeço à Aline, minha companheira, amiga, namorada e confidente, que esteve ao meu lado mesmo nos momentos mais difíceis durante esta caminhada.

Um agradecimento especial aos meus irmãos Rosineide, Cleber e Fábio, que abriram o caminho para os estudos em minha família, encorajando-me a seguir esta trajetória.

Agradeço aos colegas e professores, que contribuíram de diversas formas ao longo dos dois anos de mestrado.

Obrigado aos colegas de trabalho da COGETI/UTFPR, que proporcionaram diversas colaborações para o desenvolvimento desta dissertação.

Agradeço a todos os amigos que direta ou indiretamente contribuíram para o desenvolvimento deste trabalho.

Por fim, agradeço a meu orientador, professor Anderson Faustino da Silva, pelo rigor, pela compreensão e pelos vários ensinamentos proporcionados durante esta oportunidade de trabalharmos juntos.

Estratégias para Exploração de Sequências de Transformações do Compilador

RESUMO

Os compiladores têm por função traduzir um programa em uma linguagem fonte para uma linguagem alvo, geralmente uma linguagem de máquina. Nessa tradução, encontrar a melhor correspondência entre as linguagens é um problema complexo, pelo tamanho do espaço de busca. Por tal complexidade, uma etapa de transformação de código é necessária, na qual algoritmos de transformação modificam o código tentando melhorá-lo sem alterar seu significado. O Problema de Seleção de Transformações (PST) consiste na busca das melhores transformações para um código de entrada, tal que o código final obtenha um bom desempenho. O estado-da-arte não possui estratégias que possibilitem soluções para o PST aplicáveis a usuários finais, pois o tempo de resposta é alto para tal aplicação. O objetivo deste trabalho é formular técnicas para encontrar efetivas sequências de transformações a serem aplicadas a um código de entrada, de forma a aumentar seu desempenho reduzindo o tempo de execução. Além disso, objetiva-se reduzir o tempo de resposta de forma que a solução para o PST se aproxime da utilização por usuários finais. Inicialmente, se explora a *Variable Neighborhood Search* (VNS) para solucionar o PST, compilando iterativamente cada código de entrada. A aplicação da VNS alcançou resultados até 15,72% melhores do que outra estratégia iterativa, conseguindo melhoria em todos os programas avaliados em relação ao melhor nível de transformação. Contudo, a compilação iterativa possui alto tempo de resposta. Assim, é necessário explorar técnicas de aprendizagem de máquina, que podem prover bons resultados baseadas em experiências anteriores do compilador. Dessa forma, esta dissertação explora diferentes caracterizações de programas para representar o conhecimento acumulado na aplicação de transformações, para então aplicar a um sistema de geração de código com Raciocínio Baseado em Casos (RBC), que escolhe determinada sequência para um programa de entrada. A representação do conhecimento é capaz de atingir 81% de proximidade do melhor resultado possível para os programas avaliados, enquanto o sistema RBC gera resultados 13,74% melhores do que o nível -O3, em um tempo de resposta 99% inferior ao de estratégias de compilação iterativa. A melhoria nas formas de recuperação de experiências anteriores conseguiu superar em 20,23% o desempenho obtido por outra estratégia comparada com um número de avaliações próximo.

Palavras-chave: Compiladores. Transformações de Código. Caracterização de Programas. Raciocínio Baseado em Casos. Compilação Iterativa.

Strategies to Exploration of Compiler Transformations Sequences

ABSTRACT

Compilers aims to translate a source language program to a target language, usually a machine language. Find the best correspondence between programming languages is a complex problem, due to the size of search space. Because of this complexity, a code transformation step is needed, where transformation algorithms modify the code, trying to improve it without semantic alterations. Despite, the result of the application of these algorithms depends on code particularities. The Transformations Selection Problem (TSP) consists of the search for the best transformations to an input code, such that target code achieves a good performance. The state-of-art does not have strategies that allow the application of TSP solutions to final users, because the response time is very high to this. This work aims to formulate techniques to find effective transformations sequences to be applied to an input code, in a way to increase programs performance reducing its execution time. Furthermore, another objective is to reduce the compiler response time, to approximate a TSP solution to final users application. Initially, the metaheuristic Variable Neighborhood Search (VNS) is explored to solve TSP, compiling iteratively each input code. VNS achieved results up to 15.72% better than another iterative strategy, reaching improvement for all evaluated programs compared to the best compiler transformation level. However, the iterative compilation demands a high response time. Therefore, it is necessary to explore machine learning techniques, which can provide good results based on compiler previous experiences, with a cost of an initial training phase. Thus, this work explores different program characterizations to represent the cumulated knowledge on transformations application, to formulate a code generation system with Case-Based Reasoning (CBR), which chooses certain sequence to an input program. The knowledge representation is capable to reach 81% of proximity of the best possible result to evaluated programs, while the CBR system generates results 13.74% better than -O3 level, on a response time 99% lower than iterative compilation techniques. The improvement of previous experiences recovering method obtained performances over 20.23% compared to another strategy with a close evaluations number.

Keywords: Compiler. Code Transformations. Program Characterization. Case-Based Reasoning. Iterative Compilation.

LISTA DE FIGURAS

1.1	Exemplo de aplicação da movimentação de código invariante de laço.	12
2.1	Melhoria alcançada em cada algoritmo, em relação ao <i>baseline</i> . . .	25
2.2	Avaliações necessárias em cada algoritmo.	27
3.1	Desempenho sobre uma compilação sem a utilização de transformações.	36
3.2	<i>Speedups</i> obtidos pelo SAGC proposto para os programas teste recuperando 1, 3, 5, e 10 casos anteriores.	47
3.3	<i>Speedups</i> alcançados comparados com outras técnicas.	49
4.1	O sistema de geração de código baseado em RBC.	58
4.2	Aprendizagem em segundo plano, transparente ao usuário final. .	61
4.3	<i>Speedups</i> obtidos para cada modelo de recuperação.	63
4.4	Número de experiências a serem recuperadas para alcançar o desempenho de BestALL	68
4.5	Média das distâncias para BestALL conforme o número de casos anteriores recuperados.	70
4.6	<i>Speedups</i> obtidos aplicando as estratégias comparadas	73
4.7	Diferença entre os desempenhos de RBC-UNICO e de BestALL. . . .	75
4.8	Distância média para o desempenho de BestALL.	76
4.9	Speedup obtido pelo RBC com retroalimentação.	77

LISTA DE TABELAS

2.1	Transformações do nível -O2 da infraestrutura LLVM 3.4.	23
2.2	Parâmetros utilizados para a metaheurística VNS.	23
2.3	Tempos de execução de CE para os programas avaliados (hh:mm:ss).	24
2.4	Enumeração de semelhanças e diferenças entre a proposta apresentada e os trabalhos relacionados.	29
3.1	<i>Performance Counters</i> utilizados neste trabalho.	33
3.2	Dados de Compilação utilizados neste trabalho.	33
3.3	Características Numéricas utilizadas neste trabalho.	34
3.4	Codificação do DNA.	35
3.5	Comportamento de P_x , quando compilado com as sequências de S_0 à S_n	38
3.6	Transformações da infraestrutura LLVM 3.7.1 utilizadas.	39
3.7	Programas treino.	40
3.8	bMCoef	42
3.9	Resultados obtidos para cada programa de cBench e Polybench	44
3.10	Comparação dos trabalhos relacionados com a proposta apresentada.	53
4.1	Programas treino para construção da base dos modelos RBC.	57
4.2	Programas teste utilizados na avaliação dos modelos RBC.	62
4.3	Resultados do sistema de geração de código comparados a BestALL	66
4.4	Programas do SPEC CPU2006 utilizados neste experimento.	73

LISTA DE SIGLAS E ABREVIATURAS

AG: Algoritmo Genético
cBench: *Collective Benchmark*
CE: *Combined Elimination*
CLANG: *C Language Family Frontend for LLVM*
CN: Características Numéricas
CO: Cosseno
CPT: Representação de programas completos
CPU: *Central Process Unit*
DC: Dados de Compilação
DNA: *Deoxyribonucleic Acid*
EU: Euclidiano
GB: Gigabytes
GCC: *GNU Compiler Collection*
GHz: Gigahertz
JA: *Jaccard*
LLVM: *Low Level Virtual Machine*
MGS: Média Geométrica de Speedups
MLH: Melhoria em relação ao nível -O3
NPS: Número de Programas com desempenho Sobre -O3
NS: Número de Sequências avaliadas
NW: *Needleman-Wunsch*
PAPI: *Performance Application Programming Interface*
PC: *Performance Counters*
Polybench: Polyhedral Benchmark Suite
PST: Problema de Seleção de Transformações
QUT: Representação de programas por função quente
RAM: *Random Access Memory*
RBC: Raciocínio Baseado em Casos
SAGC: Sistema Automático de Geração de Código
SPEC: *Standard Performance Evaluation Corporation*
SVM: *Support Vector Machine*
TR: Tempo de Resposta
VNS: *Variable Neighborhood Search*

SUMÁRIO

1	Introdução	11
1.1	Motivação	12
1.2	Objetivos	13
1.3	Proposta	14
1.4	Estrutura do Trabalho	15
2	Sistema de Geração de Código por Metaheurística	16
2.1	Selecionando transformações por meio da VNS	17
2.1.1	Operadores de Vizinhança	18
2.1.2	As Rotinas de Perturbação e Busca Local	18
2.1.3	Algoritmo VNS para Encontrar Sequências de Transformações	21
2.2	Ambiente Experimental	21
2.3	Resultados Experimentais	24
2.3.1	Melhoria de Desempenho	24
2.3.2	Número de Avaliações	27
2.4	Trabalhos Relacionados	28
2.5	Considerações Finais	30
3	Representações do Conhecimento para Caracterização de Programas	31
3.1	Caracterização de Programas	32
3.2	Sistema de Geração de Código	35
3.3	Reações	35
3.3.1	Coeficientes para Identificar Reações Similares	37
3.3.2	Metodologia para Encontrar o Melhor Coeficiente	38
3.4	Base de Sequências de Transformações	39
3.5	Avaliando Representações de Conhecimento	41
3.5.1	Ambiente Experimental	41
3.5.2	Os Melhores Resultados	42
3.5.3	Resultados e Discussão	43
3.6	Sistema Automático de Geração de Código	45
3.6.1	Fase <i>Offline</i>	45
3.6.2	Fase <i>Online</i>	45
3.6.3	Metodologia	46
3.6.4	Visão Geral	46
3.6.5	Comparação	49

3.7	Trabalhos Relacionados	51
3.8	Considerações Finais	53
4	Estratégias para Seleção de Experiências Anteriores	54
4.1	O Paradigma RBC	55
4.2	Modelos para Recuperação de Sequências	55
4.3	Instanciação do Sistema	56
4.3.1	Fase <i>Offline</i>	56
4.3.2	Fase <i>Online</i>	58
4.3.3	Aprendizagem Contínua	60
4.3.4	Aprendizagem em Segundo Plano	60
4.4	Configuração Experimental	61
4.4.1	Ambiente	62
4.4.2	Metodologia	62
4.5	Resultados Experimentais	63
4.6	Melhores Soluções da Base	65
4.6.1	Número de casos	67
4.6.2	Média das Distâncias	69
4.7	Avaliação do Melhor Modelo	72
4.7.1	Ambiente Experimental	72
4.7.2	Metodologia	73
4.7.3	Resultados e Discussão	73
4.7.4	Comparação com as Melhores Soluções da Base	74
4.7.5	Uso de Aprendizagem Contínua e em Segundo Plano	77
4.8	Trabalhos Relacionados	79
4.9	Considerações Finais	79
5	Conclusão	81
5.1	Trabalhos Futuros	82
5.2	Considerações Finais	84
	REFERÊNCIAS	85

Introdução

Computação consiste no processamento de entradas, provendo soluções por meio de saídas. Para possibilitar tal processamento, combinações de instruções de máquina permitem especificar a computação a ser realizada. Porém, combinar instruções é uma tarefa complexa e geralmente requer conhecimento avançado da arquitetura do computador.

Diante disso, surgiram as linguagens de programação para servir como notação intermediária entre homem e computador (Aho et al., 2006).

Linguagens de programação de baixo nível são as mais próximas da linguagem de máquina, enquanto as linguagens de programação de alto nível são as mais próximas da linguagem humana.

As linguagens de alto nível podem ser portáveis a diferentes máquinas, o que permite ao usuário a concentração na lógica de programação e na aplicação, e não em especificidades da organização do computador (Forouzan e Mosharraf, 2011).

Porém, a execução das instruções na arquitetura do computador utilizado é possível somente por meio da linguagem de máquina, o que gera a necessidade de um tradutor de representações de alto nível para essa linguagem, papel que cabe aos compiladores.

Os compiladores têm por função traduzir um código de linguagem de programação de alto nível para um código em linguagem de máquina, sem alterar semanticamente o código fonte. Essa tradução é realizada em duas fases: (1) a fase de análise, que verifica a correteude léxica, sintática e semântica do código de entrada, e (2) a fase de tradução, que procura as melhores instruções na linguagem de máquina que correspondam ao código de entrada. Atualmente, os compiladores aplicam diversas transformações durante a fase de tradução, com o objetivo de melhorar o desempenho do código final.

As transformações de código são algoritmos que modificam um código fonte tentando melhorá-lo sem alterá-lo semanticamente. Apesar de na literatura também serem chamadas de otimizações, tal termo é impróprio, pois raramente o código resultante da aplicação de transformações é de desempenho considerado ótimo (Muchnick, 1997).

A Figura 1.1 mostra como exemplo um código que possui um laço de repetição com a operação $a \leftarrow b + c$ (na linha 3), sem que as variáveis a , b e c sejam modificadas por alguma outra instrução dentro do laço de repetição. A operação da linha 3 é considerada como invariante do laço, e a aplicação da transformação de movimentação de código invariante de laço retira tal instrução de dentro do laço de repetição, tentando diminuir o custo da execução do código final, conforme mostrado no código de saída. Porém, se o valor atribuído à variável n for zero, a execução da instrução que antes não era realizada, agora é feita uma vez, piorando o código. Ainda, se o valor de n for 1, o custo da execução do código de saída não sofreu alterações em relação ao código de entrada.

Código de entrada:

```

1  for  $i \leftarrow 1$  to  $n$  do
2  begin
3       $a \leftarrow b + c$ 
4      ...
5  end

```

Código de saída:

```

1   $a \leftarrow b + c$ 
2  for  $i \leftarrow 1$  to  $n$  do
3  begin
4      ...
5  end

```

Figura 1.1: Exemplo de aplicação da movimentação de código invariante de laço.

Considerando ainda outra possibilidade, de que o código de entrada não possua operações invariantes de laço, ao se aplicar a mesma transformação, não haveria modificações no código de saída. Assim, uma transformação é um algoritmo que tenta melhorar intuitivamente o código de entrada.

1.1 Motivação

Gerar o melhor código alvo possível para um dado código fonte é um problema complexo, que possui um amplo espaço de busca. Nesse contexto, a etapa da escolha das transformações a serem aplicadas durante o processo de compilação é um caminho para gerar códigos de boa qualidade. Porém, a escolha das melhores transformações para um código de entrada depende de suas particularidades, bem como de estruturas utilizadas.

O Problema de Seleção de Transformações (PST) consiste em encontrar a melhor sequência de transformações para um código fonte de forma a obter um bom desempenho no código final.

O estado-da-arte da busca pelas melhores transformações não possui técnicas cujo custo computacional seja razoável para obter respostas em tempo hábil para o usuário final. Assim, compiladores modernos, como GCC (Stallman e DeveloperCommunity, 2009) e CLANG (Lattner, 2002), não implementam escolha de transformações baseada no código de entrada. Esses compiladores possuem somente listas pré-fixadas para serem ativadas por meio de parâmetros de compilação (como `-O1`, `-O2` e `-O3`).

Técnicas de compilação iterativa (Leather et al., 2009; Pan e Eigenmann, 2006) tentam selecionar as transformações iterando sobre o código de entrada, avaliando-o a cada aplicação. O problema é o custo dessas técnicas, que necessitam da compilação e execução do código gerado a cada avaliação.

Com o intuito de reduzir o custo de técnicas iterativas, foram propostas técnicas de aprendizagem de máquina (Cavazos et al., 2007; Queiroz Junior e da Silva, 2015). Tais técnicas possuem o atrativo de fazer com que uma nova experiência seja conhecimento acumulado para novas buscas, reduzindo o número de avaliações realizadas.

1.2 Objetivos

O objetivo deste trabalho é formular técnicas para encontrar efetivas sequências de transformações, de forma a aumentar o desempenho de programas reduzindo o seu tempo de execução. Além disso, outro objetivo é reduzir o tempo de resposta do compilador de forma que uma solução para o PST se aproxime da aplicação a usuários finais.

Para alcançar tal objetivo, objetivos específicos são estabelecidos:

- Aplicar uma técnica de compilação iterativa para solucionar o problema, de forma a avaliar o desempenho obtido e o tempo de resposta;
- Formalizar uma estratégia para avaliar representações do conhecimento, que são utilizadas para comparar experiências e possibilitar aplicação de transformações similares a programas similares;
- Aplicar uma técnica de aprendizagem de máquina para solucionar o problema, capaz de reutilizar experiências anteriores que utilizam a representação formalizada;
- Avaliar o uso de diferentes estratégias para obter experiências anteriores em um sistema de geração de código;

- Formalizar uma estratégia capaz de vincular transformações a programas.

1.3 Proposta

A aplicação de metaheurísticas é um caminho viável para o desenvolvimento de técnicas de compilação iterativa. Dessa forma, esta dissertação aplica a metaheurística *Variable Neighborhood Search* (VNS) à seleção de transformações de código.

A avaliação experimental evidenciou que os operadores de busca local definidos para a metaheurística utilizada são eficientes na seleção de transformações. Além disso, comparado à técnica *Combined Elimination* (Pan e Eigenmann, 2006), o algoritmo proposto conseguiu resultados até 15,72% melhores com uma quantidade menor de avaliações.

A compilação iterativa pode prover bons resultados, porém o tempo de resposta é alto e isso não é aplicável a usuários comuns de compiladores. Técnicas de aprendizagem de máquina podem fornecer bons resultados com um tempo de resposta menor do que o necessário em técnicas de compilação iterativa. Contudo, ao custo de uma fase inicial de treinamento.

Desta forma, esta dissertação também avalia a aplicação do *Raciocínio Baseado em Casos* (RBC) para o mesmo problema. Além disso, exploram-se distintas representações do conhecimento e estratégias para obter experiências anteriores.

Por fim, da experiência adquirida até o momento, esta dissertação propõe uma estratégia capaz de escolher determinada sequência de transformações para um programa específico, o qual é representado por um formalismo capaz de capturar suas principais características. Além disso, um sistema RBC que utiliza tal representação é capaz de obter resultados superiores aos obtidos por outras estratégias.

Quanto à representação de conhecimento, as avaliações mostraram que a caracterização utilizada é capaz de atingir 81% de proximidade do melhor resultado possível, enquanto o sistema RBC é capaz de gerar resultados 13,74% melhores do que o nível mais agressivo de transformações da infraestrutura de compilação LLVM (Lattner, 2002), realizando somente 10 avaliações e com tempo de resposta 99% inferior ao de técnicas de compilação iterativa.

Avaliando modelos de recuperação de experiências anteriores, foi possível alterar o sistema RBC de forma a gerar resultados 20,23% superiores à técnica *Best10* (Purini e Jain, 2013), obtendo cobertura de 66,67% em relação ao nível -03 da LLVM para os *benchmarks* do SPEC CPU2006.

1.4 Estrutura do Trabalho

Os capítulos desta dissertação são auto-contidos. Tal organização possui dois objetivos: (1) demonstrar como ocorreu o progresso do trabalho; e (2) facilitar a leitura de cada capítulo, tornando cada capítulo independente dos outros. Além disso, cada um dos três capítulos do desenvolvimento do trabalho resultou em uma publicação diferente.

O Capítulo 2 descreve a proposta da aplicação de metaheurística à compilação iterativa, bem como sua avaliação experimental e resultados alcançados. Esse capítulo foi publicado no XLVII Simpósio Brasileiro de Pesquisa Operacional, com o título *Uma Estratégia Baseada na Metaheurística VNS para Encontrar Efetivas Sequências de Otimizações*.

O Capítulo 3 apresenta as representações de conhecimento para caracterização de programas e formaliza um meio de avaliá-las por reações à aplicação de transformações. Tal capítulo ainda aplica um sistema de geração de código para demonstrar o uso de tal representação, expondo resultados e comparando com outras estratégias. A publicação resultante desse capítulo foi aceita para a 19th International Conference on Enterprise Information Systems, com o título *Evaluating Knowledge Representations for Program Characterization*.

O Capítulo 4 vai mais a fundo no sistema de geração de código, explorando modelos de recuperação de experiências anteriores, mostrando resultados e avaliações de tal sistema, comparando-o com outra estratégia e com as melhores soluções possíveis da base de conhecimento. Até a data de entrega desta dissertação, esse capítulo ainda não havia sido publicado, contudo está em processo de preparação para um formato de artigo a ser submetido.

Por fim, o capítulo 5 apresenta as conclusões desta dissertação, expondo resultados alcançados e sugerindo trabalhos futuros.

Sistema de Geração de Código por Metaheurística

Na tentativa de procurar soluções viáveis para a geração de código, várias abordagens foram propostas na literatura. Dentre essas abordagens estão: busca exaustiva (Foleiss et al., 2011), técnica estatística (Haneda et al., 2005), eliminação iterativa (Pan e Eigenmann, 2006), algoritmos genéticos (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 (Purini e Jain, 2013).

Utilizando uma abordagem diferente, uma das propostas desta dissertação é o uso da clássica metaheurística *Variable Neighborhood Search* (VNS) (Mladenović e Hansen, 1997), para mitigar o Problema de Seleção de Transformações (PST) considerando a busca por sequências de transformações com repetição.

Esse tipo de busca foi escolhido por ter o espaço mais amplo. Outras formas de tratamento do problema são: (1) busca por conjunto e (2) busca por sequência sem repetição. A busca por conjunto se limita a explorar a presença ou ausência de cada transformação com uma ordem de aplicação pré-definida. A busca por sequência sem repetição considera diferentes formas de ordenação na aplicação das transformações. A busca por sequência com repetição ainda pode aplicar cada transformação mais de uma vez, tendo assim um espaço de busca (teoricamente) infinito.

A estratégia implementada mostrou ser eficiente para encontrar boas sequências para os *benchmarks* do SPEC CPU2006 (Henning, 2006). Além disso, a estratégia proposta se mostrou mais atrativa do que *Combined Elimination* (CE) (Pan e Eigenmann, 2006).

2.1 Selecionando transformações por meio da VNS

O algoritmo VNS (Mladenović e Hansen, 1997) é uma metaheurística utilizada na aplicação de problemas combinatórios e de otimização global. Sua proposta é efetuar sistematicamente mudanças na vizinhança de uma solução com uma busca local para encontrar novas soluções.

O algoritmo possui uma abordagem melhorativa, e assim possui como entrada uma solução inicial. Além disso, uma lista de operadores é necessária para explorar as vizinhanças de uma solução.

Um algoritmo que utiliza a metaheurística VNS segue o comportamento mostrado no Algoritmo 1.

Algoritmo 1: Comportamento da metaheurística VNS

```

Input: Solução inicial (I)
          Lista de operadores (lista_operadores)
Output: Solução final (S)
 $S \leftarrow I$ 
while Critério de parada não satisfeito do
   $n \leftarrow 1$ 
  while  $n \leq |lista\_operadores|$  do
     $S' \leftarrow Pertubar(S, lista\_operadores[n])$ 
     $S'' \leftarrow Busca\_Local(S', lista\_operadores[n])$ 
    if  $S''$  é melhor do que  $S$  then
       $S \leftarrow S''$ 
    else
       $n \leftarrow n + 1$ 
  return  $S$ 

```

O trabalho de Lima (2013) propôs a aplicação dessa metaheurística ao PST, sugerindo e avaliando um modelo de implementação. Partindo desse modelo, uma das propostas desta dissertação é expandir o espaço explorado, efetuando uma busca que se adapte às soluções encontradas. Para possibilitar tal expansão, taxas de exploração definem a quantidade de buscas nas vizinhanças. Além disso, o algoritmo foi flexibilizado se adaptar ao programa de entrada por meio de novos parâmetros e critérios de parada.

Considerando tanto repetições quanto ordem, o uso da metaheurística VNS sugere ser uma boa abordagem para melhorar a geração de código por meio da busca de sequências de transformações.

2.1.1 Operadores de Vizinhança

Operadores de vizinhança são métodos que modificam uma solução para buscar soluções vizinhas à ela, seguindo critérios pré-determinados, e realizam a principal operação da busca local. No PST, o espaço de busca é ampliado ao explorar não só quais transformações estarão mas também em que ordem elas estarão. Além disso, é possível repetir a aplicação de uma mesma transformação. Os operadores escolhidos são os mesmos do trabalho de Lima (2013):

Troca Permuta a ordem de duas transformações aleatórias;

Remoção escolhe uma transformação de forma arbitrária e a remove; e

Inserção escolhe aleatoriamente uma transformação e a adiciona à sequência, em uma posição aleatória.

A ordem de aplicação é explorada pelo operador de Troca, enquanto Remoção e Inserção exploram a presença ou ausência das transformações na sequência. Não há restrições quanto a repetições das transformações nas sequências.

2.1.2 As Rotinas de Perturbação e Busca Local

A rotina de perturbação tenta expandir as regiões de busca. Se estabelece também uma taxa de perturbação para ampliar o espaço de busca conforme a reação da solução, na qual cada operador é aplicado com sua proposta de expansão ou compressão. O Algoritmo 2 apresenta a proposta de modificação na rotina de perturbação. Diferente da proposta de Lima (2013), esta rotina perturba a solução se baseando em uma taxa de perturbação, a qual é incrementada conforme haja convergência por meio de perturbações ou decrementada se não houver.

Algoritmo 2: Rotina de perturbação

Input: Solução a ser perturbada (S)
 Taxa de Perturbação (Tx)
 Benchmark (B)
Output: Solução perturbada (S'); Tempo de Execução da Solução (T)
 $lista_operadores \leftarrow [Troca, Remove, Inse]$
 $Operador \leftarrow Random(lista_operadores)$
 $Q \leftarrow |S| * Tx$
 $S' \leftarrow ApliqueOperador(Operador, S, Q)$
 $Tempo_execucao \leftarrow Avalie(S', B)$
return $S', Tempo_execucao$

Perturbar uma solução significa escolher um operador de forma aleatória e aplicá-lo a solução que deve ser perturbada, fornecendo desta forma uma nova solução.

A busca local tem por objetivo procurar a melhor solução em uma vizinhança, que pode ser realizada explorando toda a vizinhança da solução. Entende-se por toda vizinhança todas as soluções que podem ser geradas a partir de uma determinada solução com um determinado operador. Contudo, explorar todas as possibilidades gera um alto custo pois envolve a avaliação de cada código gerado¹. Dessa forma, é necessário limitar o número de soluções exploradas em cada vizinhança. Na solução proposta isso é realizado com base em uma taxa de exploração, a qual é relativa ao tamanho da melhor solução encontrada até o momento.

Seguindo tal estratégia, foram desenvolvidas três buscas locais com os operadores troca, remoção e inserção.

Algoritmo 3: Busca local com o operador “troca”

Input: Solução para a qual deve ser analisada a vizinhança (S)
 Tempo de execução da solução S (T)
 Taxa de troca (Tx)
Benchmark (B)

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

```

 $Q \leftarrow |S| \times Tx$ 
for  $i \leftarrow 1$  to  $Q$  do
   $S' \leftarrow \text{ApliqueOperador}(Troca, S, 1)$ 
   $T' \leftarrow \text{Avalie}(S', B)$ 
  if  $T' < T$  then
     $S \leftarrow S'$ 
     $T \leftarrow T'$ 
return  $S, T$ 

```

A busca local de troca efetua uma troca aleatória entre duas transformações e avalia a nova solução, conforme apresentado no Algoritmo 3. Se a troca for benéfica, então essa solução é utilizada na próxima iteração. A busca finaliza após Tx_{troca} ² trocas aleatórias. Esse operador tem por objetivo melhorar a solução explorando a ordem de aplicação das transformações.

A busca local de remoção retira uma transformação de S , gerando S' , como apresenta o Algoritmo 4. O desempenho de S' é então avaliado e, se S' tem desempenho superior a S , uma próxima remoção será realizada em S' . Dessa forma, são eliminadas da solução as transformações que afetam negativamente o desempenho, tendo o máximo de remoções possíveis definido por Tx_{remove} .

O Algoritmo 5 apresenta a busca local de inserção, que adiciona Tx_{insere} novas transformações à solução inicial. Tanto a posição de inserção quanto a própria transformação

¹Avaliar uma solução significa compilar o programa com a sequência encontrada, executá-lo e coletar seu tempo de execução.

² Tx_{troca} é a taxa de trocas, indicando a quantidade máxima de trocas que serão avaliadas.

Algoritmo 4: Busca local com o operador “remoção”

Input: Solução para a qual deve ser analisada a vizinhança (S)
 Tempo da solução S (T)
 Taxa de remoção (Tx)
 Benchmark (B)

$lista_rem \leftarrow \emptyset$
 $Q \leftarrow |S| \times Tx$
for $i \leftarrow 1$ **to** Q **do**
 $S' \leftarrow S$
 $k \leftarrow$ Número aleatório entre 1 e $tamanho(S)$ que não esteja em $lista_rem$
 $lista_rem.append(k)$
 Remove o elemento na posição k de S'
 $T' \leftarrow Avalie(S', B)$
 if $T' < T$ **then**
 $S \leftarrow S'$
 $T \leftarrow T'$
 $lista_rem \leftarrow \emptyset$

return S, T

Algoritmo 5: Busca local com o operador “inserção”

Input: Solução para a qual deve ser analisada a vizinhança (S)
 Tempo da solução S (T)
 Taxa de inserção (Tx)
 Benchmark (B)

$lista_ins \leftarrow \emptyset$
 $Q \leftarrow |S| \times Tx$
for $i \leftarrow$ **to** Q **do**
 $S' \leftarrow S$
 $k \leftarrow$ Número aleatório entre 1 e $tamanho(S)$ que não esteja em $lista_ins$
 $lista_ins.append(k)$
 $o' \leftarrow Random(O)$ /* O é uma constante que contém todas as transformações disponíveis */
 Insira o' na posição k de S_i /*sem sobrepor transformação*/
 $T' \leftarrow Avalie(S', B)$
 if $T' < T$ **then**
 $S \leftarrow S'$
 $T \leftarrow T'$
 $lista_ins \leftarrow \emptyset$

return S, T

a ser inserida são aleatórias. Isso permite a inserção de transformações em diferentes posições, além da repetição de aplicação da mesma transformação.

O uso de diferentes buscas locais amplia o espaço de busca, fornecendo novas possibilidades para encontrar boas soluções. Além disso, a utilização de taxas para cada operador possibilita encontrar soluções em um espaço de busca mais amplo, pois, quando as tentativas não encontram uma solução melhor, as taxas alteram o comportamento do algoritmo com o objetivo de ampliar o espaço de busca e assim aumentar a convergência do algoritmo para um ótimo global.

2.1.3 Algoritmo VNS para Encontrar Sequências de Transformações

Além das rotinas de perturbação, busca local e operadores de vizinhança, um algoritmo com a metaheurística VNS necessita de um critério de parada e uma solução inicial.

O algoritmo proposto finaliza a busca por novas soluções quando um dos seguintes critérios ocorre:

1. **Tempo de execução:** este critério limita o tempo de execução da metaheurística, determinando o tempo máximo no qual o sistema deve fornecer uma resposta. Este critério indica o tempo total de execução do sistema. Portanto, ele inclui: (1) tempo de perturbar uma solução, (2) tempo de buscar novas soluções, (3) tempo de compilar o *benchmark* com uma determinada solução e (3) tempo de executar o programa para validar a solução encontrada.
2. **Iterações:** sendo um algoritmo iterativo é importante definir a quantidade máxima de iterações.
3. **Avaliações:** dada a natureza do problema, este critério tem por objetivo definir a quantidade máxima de compilações que um programa sofrerá.
4. **Desempenho:** este critério define um limiar máximo ao desempenho esperado.

Tais critérios de parada possuem o atrativo de flexibilizar o ajuste da metaheurística de acordo com as características do programa em questão.

Uma questão importante é a escolha da sequência inicial. Algoritmos heurísticos podem utilizar uma abordagem construtiva, melhorativa ou ambas.

Devido à natureza da VNS, o algoritmo proposto pode ser visto como um algoritmo melhorativo. Como o objetivo final é melhorar o desempenho do melhor nível de transformações padrão do compilador em questão, o ideal é que a solução de partida seja uma solução conhecida previamente. Dessa forma, a solução de partida é a melhor sequência padrão do compilador em questão.

O Algoritmo 6 apresenta a metaheurística VNS proposta para o encontrar efetivas sequências de transformações.

2.2 Ambiente Experimental

Arquitetura A arquitetura utilizada foi um computador com processador Intel(R) Core(TM) I7-2600 CPU 3.4GHz com memória RAM de 4GB. O sistema opera-

cional executando na máquina foi o Ubuntu 13 x86_64, sob a versão de *kernel* 3.11.0-15-generic.

Programas Para avaliar a solução proposta foram utilizados os *benchmarks* do SPEC CPU2006 (Henning, 2006) implementados em C ou C++, para a entrada `train`. Isso devido outras entradas possuem alto tempo de execução, o que elevaria o tempo dos experimentos. Para evitar flutuações no tempo de execução, cada programa foi executado diversas vezes, até que a variância da amostragem de execução fosse menor do que 1.

Compilador A infraestrutura de compilação LLVM 3.4 (Lattner e Adve, 2004).

Baseline Para definir o nível de transformação que seria utilizado como *baseline* um conjunto de experimentos foi realizado com os *benchmarks* do SPEC CPU2006. Nesses experimentos o nível de transformação `-O2` obteve no geral o melhor desempenho. Dessa forma, a sequência `-O2` foi escolhida como *baseline*. Essa é também a sequência de partida para a metaheurística VNS, e as transformações presentes nela formam a

Algoritmo 6: VNS para encontrar efetivas sequências de transformações.

```

Input: Benchmark (B),
        Sequência inicial (S),
        Lista de operadores (lista_operadores)
Output: Melhor sequência encontrada (melhor_seq)
lista_taxas ← [Tx.troca, Tx.remove, Tx.insere]
k ← 0
melhor_t ← tempo_de_execucao_da_sequencia_inicial()
melhor_seq ← seq
count ← 0
convergiu ← false
while condição de parada não for satisfeita do
    while k < |lista_operadores| do
        count ← count + 1
        S, T ← perturbacao(S, B)
        /* k = 0 troca; k = 1 remove; k = 2 insere */
        S, T ← busca_local_k(S, T, lista_taxas[k], B)
        if T < melhor_t then
            melhor_t ← t
            melhor_seq ← seq
            k ← 0
            convergiu ← true
            Atribuir valores-padrão às taxas
        else
            S ← melhor_seq
            k ← k + 1
    if ((count % (|lista_operadores| × 2) = 0) & NOT(convergiu)) then
        Dobrar todos os valores de taxas
        count ← 0
    if convergiu then
        convergiu ← false
return melhor_seq

```

sequência na qual novas transformações serão buscadas. A Tabela 2.1 apresenta as transformações do nível `-O2` da LLVM 3.4.

```
-targetlibinfo -no-aa -tbaa -basicaa -notti -globalopt -ipsccp -deadargelim -instcombine
-simplifcfcg -basiccg -prune-eh -inline-cost -inline -functionattrs -sroa -domtree -early-cse
-lazy-value-info -jump-threading -correlated-propagation -simplifcfcg -instcombine
-tailcallelim -simplifcfcg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm
-lcssa -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indvars -loop-idiom
-loop-deletion -loop-unroll -memdep -gvn -memdep -memcpyopt -sccp -instcombine
-lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -adce
-simplifcfcg -instcombine -strip-dead-prototypes -globaldce -constmerge -preverify -domtree
-verify
```

Tabela 2.1: Transformações do nível `-O2` da infraestrutura LLVM 3.4.

Comparação Uma questão importante é avaliar o algoritmo proposto com algum algoritmo apresentado na literatura. O algoritmo **CE**, desenvolvido por Pan e Eigenmann (2006), foi o escolhido para comparação. Isso devido a dois fatores: (1) por **CE** pertencer à mesma classe do algoritmo proposto – compilação iterativa; (2) pela literatura apresentar bons resultados para **CE**.

Parâmetros da Metaheurística Para calibrar o algoritmo, foram realizados experimentos com cada parâmetro em uma amostragem de um *benchmark* do experimento. Assim, os melhores parâmetros encontrados para o algoritmo proposto são apresentados na Tabela 2.2.

Parâmetro	Valor	Parâmetro	Valor
Sequência inicial	<code>[-O2]</code>	Taxa de perturbação	10%
Tempo de execução	CE e 4h	Taxa de troca	40%
Iterações	25	Taxa de remoção	20%
Avaliações	5000	Taxa de inserção	10%
Desempenho	50%		

Tabela 2.2: Parâmetros utilizados para a metaheurística **VNS**.

Cenários A análise experimental compreende dois cenários diferentes:

1. O algoritmo proposto foi parametrizado com o tempo de execução do algoritmo **CE**. O primeiro cenário tem por objetivo identificar o desempenho do algoritmo proposto mediante o mesmo tempo de execução do algoritmo **CE**.
2. O algoritmo proposto foi parametrizado com o tempo de execução de 4 horas. O segundo cenário tem por objetivo padronizar a execução do algoritmo independente do programa em questão.

O tempo de execução do algoritmo **CE** é apresentado na Tabela 2.3.

Programa	Tempo de execução	Programa	Tempo de execução
400.perlbenc (S00)	4:14:25.771	456.hmmmer (S10)	8:33:15.959
401.bzip2 (S01)	8:03:37.629	458.sjeng (S11)	18:37:17.695
403.gcc (S02)	2:54:07.430	462.libquantum (S12)	0:19:29.499
429.mcf (S03)	3:18:52.776	464.h264ref (S13)	8:59:54.149
433.milc (S04)	0:42:21.966	470.lbm (S14)	3:29:05.226
444.namd (S05)	2:10:15.865	471.omnetpp (S15)	14:05:05.937
445.gobmk (S06)	21:51:54.263	473.astar (S16)	16:29:09.700
447.dealIII (S07)	5:15:19.524	482.sphinx3 (S17)	1:57:18.804
450.soplex (S08)	1:45:17.497	483.xalancbmk (S18)	28:19:00.061
453.povray (S09)	2:53:19.272		

Tabela 2.3: Tempos de execução de CE para os programas avaliados (hh:mm:ss).

A Metaheurística VNS O algoritmo proposto, por ser baseado em uma metaheurística, pode apresentar diferentes respostas em execuções distintas. Tradicionalmente, um algoritmo heurístico é executado N vezes e o resultado final computado a partir dessas execuções. A abordagem utilizada para avaliar o algoritmo proposto foi executá-lo 3 vezes e para cada programa e apresentar o resultado de cada execução.

2.3 Resultados Experimentais

A avaliação experimental é norteadada por duas métricas: (1) melhoria de desempenho e (2) número de avaliações. A primeira indica o percentual de melhoria alcançado por um determinado algoritmo, em relação ao *baseline*. A segunda indica a quantidade de compilações necessárias para o programa em questão.

2.3.1 Melhoria de Desempenho

A Figura 2.1 apresenta a melhoria alcançada em cada algoritmo. Para cada *benchmark* são apresentadas 7 barras na seguinte ordem da esquerda para a direita: CE, VNS(1), VNS(2), VNS(3), VNS(4H.1), VNS(4H.2) e VNS(4H.3). VNS(X) é a configuração do primeiro cenário, e VNS(4H. X) a do segundo cenário.

Cenário 1

Os resultados indicam que a estratégia utilizada para mitigar a escolha de transformações é melhor do que a estratégia utilizada por CE. Enquanto CE tem como premissa remover as transformações prejudiciais, o algoritmo proposto parte da premissa que um ótimo global será encontrado à medida em que a sequência de transformações possa ser expandida ou reduzida. Outro ponto importante, é o fato de CE mitigar apenas soluções pré-estabelecidas

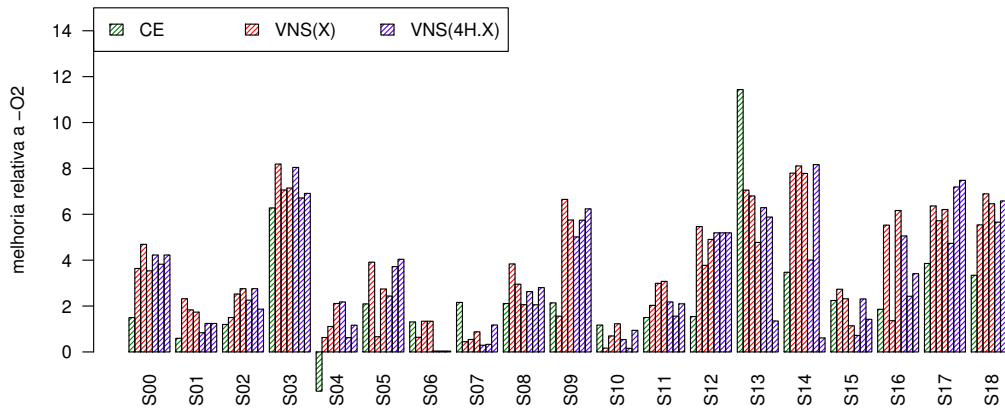


Figura 2.1: Melhoria alcançada em cada algoritmo, em relação ao *baseline*.

sem modificar sua ordem, enquanto o algoritmo proposto por meio do operador de trocas, tenta mitigar em conjunto o problema da ordenação de transformações.

Apenas para dois programas (S07 e S13) a metaheurística não superou CE. Para os outros programas pelo menos uma execução superou CE, sendo que para 10 programas todas execuções superaram CE.

O *speedup* médio³ de CE é de 1,023, enquanto o do algoritmo proposto é de 1,036. Sendo de 1,036, 1,035 e 1,038, para VNS(1), VNS(2), VNS(3), respectivamente.

Uma análise da melhoria do algoritmo proposto (valor médio entre as três execuções) indica que este é capaz de alcançar um desempenho superior a CE que varia entre 15,71% (S03) e 250,38% (S04), desconsiderando os programas para os quais houve perda de desempenho (que variou entre -8,89% e -261,85%, para os programas S05, S06, S07, S10, S13 e S15).

Cenário 2

Esta configuração pode beneficiar alguns programas, enquanto reduz as chances da metaheurística encontrar um ótimo global para outros. Para o *benchmark* S00, o tempo de execução do algoritmo proposto foi praticamente o mesmo do utilizado no cenário 1. Por outro lado, para os programas S01, S06, S07, S10, S11, S13, S15, S16 e S18 houve uma redução no tempo de execução, enquanto para o restante houve aumento.

³Os valores médios se referem à média geométrica.

Esse fator possui prós e contras. Tradicionalmente, quanto maior o tempo de execução de um algoritmo heurístico maior é a probabilidade de ser encontrado um ótimo global e não um ótimo local. Por outro lado, um tempo de execução elevado não garante necessariamente que um ótimo global será encontrado, pela natureza do algoritmo. Por isso é necessário ter vários critérios de parada, além de um auto-ajuste nas características do algoritmo. No algoritmo proposto, existem 4 critérios de parada, além do auto-ajuste das taxas dos operadores.

Em uma análise geral, neste segundo cenário, o algoritmo proposto não superou CE para S06 e S10, além dos dois programas do cenário 1. Contudo, nesta nova configuração o algoritmo proposto superou CE para 12 programas, em todas as execuções.

O *speedup* médio obtido pelo algoritmo proposto é de 1,033. Sendo de 1,033, 1,035 e 1,030, para VNS(4H.1), VNS(4H.2), VNS(4H.3), respectivamente.

Uma análise da melhoria do algoritmo proposto (valor médio entre as três execuções, como no cenário anterior) indica que o cenário 2 tem desempenho similar ao cenário 1 no tocante a quantidade de programas para os quais o algoritmo proposto obteve uma melhor solução do que CE. Quanto ao ganho, este variou entre 12,80% e 246,81%. Enquanto a perda variou entre -28,07% e -3499,73%, para os programas S06, S07, S10, S13, S14 e S15.

Observando o desempenho do algoritmo proposto é possível perceber que a redução (ou aumento) do tempo de execução impactou o percentual de melhoria. Para todos os programas que tiveram seu tempo de execução reduzido houve uma queda de desempenho comparado tanto com a execução do cenário 1, conseqüentemente uma perda do ganho sobre CE. Embora, neste último caso, a melhor opção ainda seja utilizar o algoritmo proposto.

Para 5 programas (S02, S05, S09, S12 e S17) o aumento do tempo de execução resultou em um ganho de desempenho. É importante perceber que o aumento do tempo de execução não é proporcional ao ganho obtido. Para S05 o tempo de execução dobrou e o ganho em relação a CE, que para um tempo de execução maior foi de 1,923%, passou para 3,319%. Para S12 o tempo de execução aumentou em um fator de 12 vezes, contudo o ganho passou de 4,649% para 5,191%. Além disso, para S03, S08 e S14 houve uma perda de desempenho para um tempo de execução maior.

Em síntese os resultados indicam que a melhor abordagem é aumentar o tempo de execução do algoritmo. Além disso, a análise detalhada dos dados indica que a estratégia de auto-ajuste das taxas dos operadores é uma boa estratégia para garantir que o algoritmo continue convergindo por um longo período. Para todos os programas,

o algoritmo terminou por atingir o tempo de execução máximo. Além disso, para todos foram encontradas soluções parciais.

2.3.2 Número de Avaliações

A Figura 2.2 apresenta a quantidade de avaliações de cada algoritmo. Como a Figura 2.1, para cada *benchmark* são apresentados 7 barras na seguinte ordem da esquerda para a direita: CE, VNS(1), VNS(2), VNS(3), VNS(4H.1), VNS(4H.2) e VNS(4H.3). VNS(x) é a configuração do primeiro cenário, e VNS(4H. x) a do segundo cenário.

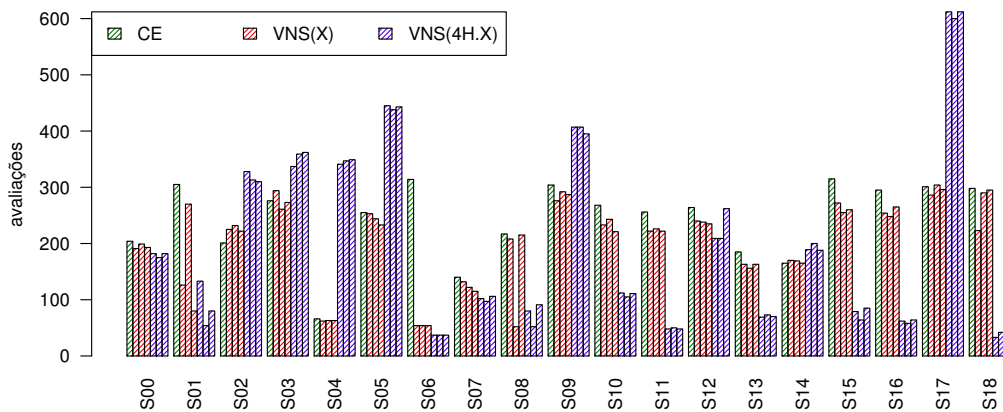


Figura 2.2: Avaliações necessárias em cada algoritmo.

Como esperado a quantidade de avaliações necessárias por VNS(x) é similar a quantidade necessária por CE. Isso por ambos os algoritmos terem o mesmo tempo de execução.

Para alguns programas existe uma variação nessa quantidade, que pode ser explicada pela natureza da sequência gerada pelo algoritmo proposto. Em geral, o CE avalia transformação por transformação para determinar se ela é nociva ao desempenho do *benchmark* em questão. Como o algoritmo proposto, além de remover, investe em trocas e inserções, converge mais rápido para uma melhor solução. Conseqüentemente, reduz o tempo de execução do programa proporcionando que novas avaliações sejam realizadas.

Um ponto importante a ser ressaltado é o fato de ser possível obter um ganho de desempenho sobre um nível de transformação do compilador em questão, com uma quantidade menor de avaliações do que a necessária por CE. Isso é um ponto importante, pois assim como CE o algoritmo proposto é um algoritmo de compilação iterativa. Dessa

forma, quanto menor a quantidade de avaliações necessárias menor será o tempo de resposta do sistema.

2.4 Trabalhos Relacionados

O trabalho de Foleiss et al. (2011) utilizou busca exaustiva em algumas classes específicas de transformações, e não para cada transformação isoladamente. Essa restrição permitiu que fosse viável a busca exaustiva no problema apresentado. Diferentemente, o trabalho proposto no presente trabalho avalia cada transformação individualmente. Além disso, enquanto o trabalho de Foleiss et al. (2011) tem por objetivo tamanho de código, o presente trabalho tem como objetivo tempo de execução.

Técnicas estatísticas foram utilizadas no trabalho de Haneda et al. (2005), no qual o teste de hipótese de Mann-Whitney serviu para verificar se determinada transformação afeta ou não o código gerado, permitindo assim decidir se é viável selecioná-la ou não. O trabalho de Haneda et al. (2005) se difere do proposto neste trabalho, pelo fato de este não utilizar técnicas estatísticas e sim um algoritmo heurístico de busca.

O trabalho de Pan e Eigenmann (2006) utilizou eliminação iterativa para selecionar boas sequências de transformações. O objetivo foi investigar quais transformações prejudicavam a qualidade para determinado código e então retirá-las da sequência padrão. Enquanto o trabalho citado apenas remove as transformações prejudiciais, o presente trabalho propõe o uso de inserção e troca de transformações, além da remoção. Dessa forma, o presente trabalho tenta explorar um espaço de busca maior do que aquele explorado pelo algoritmo de Pan e Eigenmann (2006).

Abordagens genéticas foram utilizadas tanto para busca de boas sequências de transformações em termos de *speedup* e tamanho de código, quanto para construções de heurísticas a serem utilizadas em transformações específicas como, por exemplo, para decidir o melhor nível de aplicação das transformações *loop unroll* (Leather et al., 2009). O algoritmo proposto, que é baseado na VNS, possui similaridade com tais trabalhos pelo fato de ambos utilizarem uma abordagem heurística.

O trabalho de Lau et al. (2006) combinou aleatoriedade e técnicas estatísticas para determinar quais transformações são boas para uma determinada região de código no contexto das máquinas virtuais (Smith e Nair, 2005) e compilação *just-in-time*. O algoritmo heurístico proposto no presente trabalho aplica transformações a todo o código, e não à regiões específicas e, portanto, todas as transformações aplicadas são consideradas para o todo do programa.

Uma abordagem mais recente para o problema de geração de código (Purini e Jain, 2013) combinou pelo menos quatro técnicas diferentes para seleção de transformações, a saber: busca aleatória, algoritmos genéticos, aprendizagem de máquina e eliminação iterativa. Ao aplicar tais técnicas, os autores reduziram o espaço de busca às poucas sequências de transformações encontradas, as quais são capazes de alcançar desempenho para um grande número de programas. A principal diferença com a proposta apresentada é o tipo de abordagem, pois enquanto o trabalho de Purini e Jain (2013) reduz a busca a poucas sequências encontradas, o algoritmo proposto parte de uma solução a fim de melhorá-la por meio da compilação iterativa.

O trabalho de Lima (2013) propôs, entre outras técnicas, uma aplicação da metaheurística VNS ao PST, definindo os operadores utilizados nesta dissertação. Contudo, diferentemente da proposta desta dissertação, o trabalho de Lima (2013) não flexibiliza o critério de parada do algoritmo, iterando um número de vezes fixado para o programa de entrada. Além disso, a proposta de Lima (2013) não amplia o espaço de busca por taxas de perturbação ou busca local, explorando um limite fixo de soluções vizinhas.

A Tabela 2.4 apresenta a lista de trabalhos relacionados e sua comparação com a proposta de aplicação da VNS ao PST.

Tabela 2.4: Enumeração de semelhanças e diferenças entre a proposta apresentada e os trabalhos relacionados.

Trabalho	Estratégia	Semelhanças	Diferenças
Foleiss et al. (2011)	Busca Exaustiva	Aplicação ao PST	Objetivo de tamanho de código
Haneda et al. (2005)	Estatística	Aplicação ao PST	Busca estatística
Pan e Eigenmann (2006)	Eliminação Iterativa	Abordagem melhorativa	Exploração somente da remoção
Lau et al. (2006)	Aleatória e Estatística	Aplicação ao PST	Aplicação em compilação JIT
Leather et al. (2009)	Heurística	Aplicação de metaheurística	A metaheurística foi um Algoritmo Genético
Purini e Jain (2013)	Várias	Compilação Iterativa	Reduz o espaço de busca a poucas sequências
Lima (2013)	Heurística	Metaheurística VNS	Flexibilidade dos parâmetros e taxas de exploração

2.5 Considerações Finais

Dada a necessidade de produzir códigos com mais qualidade, os projetistas de compiladores implementam dezenas de transformações. No entanto, a complexidade da relação entre as diferentes transformações torna complexo determinar quais garantirão um bom desempenho. Dessa forma, é necessário avaliar diversas possibilidades. Contudo, a exploração de todas as possibilidades é inviável devido ao tamanho do espaço de busca.

A abordagem proposta neste trabalho para mitigar o problema em questão permitiu a exploração de um amplo espaço de busca, apresentando bons resultados. A utilização de taxas de exploração proporcionou a maximização do espaço de busca, permitindo que em uma possível estagnação do algoritmo a busca se concentrasse em outro ponto do espaço.

A avaliação experimental indica as contribuições deste trabalho. Os resultados evidenciaram que os operadores de busca local definidos para a metaheurística utilizada são adequados à seleção de transformações. Para todos os experimentos realizados, não houve sequer um programa avaliado para o qual o algoritmo proposto não conseguisse alguma melhoria. O algoritmo proposto mostrou que é possível obter resultados melhores do que aqueles obtidos por CE, mesmo com uma quantidade menor de avaliações.

Apesar dos bons resultados, a estratégia apresentada neste capítulo demanda alto tempo de execução, o que é impraticável na aplicação a usuários finais.

Sugere-se como trabalho futuro a aplicação de uma estratégia de aprendizagem de máquina que forneça respostas em um menor tempo. Estratégias de aprendizagem de máquina necessitam de um formalismo para extrair conhecimento do contexto em que são aplicadas. Dessa forma, existe a necessidade de uma especificação de uma representação de conhecimento para identificar programas similares, que possibilite a aplicação da aprendizagem de máquina a fim de reconhecer boas sequências de transformações para programas similares.

Representações do Conhecimento para Caracterização de Programas

Representação do conhecimento é um campo da inteligência artificial dedicado a representar o que se sabe sobre um contexto específico como o intuito de ser utilizado para criar formalismos e, assim, resolver problemas complexos.

Um problema complexo, no contexto da ciência da computação, é a geração de bons códigos por compiladores, devido principalmente às características do programa fonte.

A geração de código passa por uma etapa de seleção de transformações, na qual algoritmos específicos tentam melhorar o código de entrada de forma intuitiva. Porém, as características específicas de cada programa demandam diferentes sequências de transformações aplicadas. Dessa forma, o Problema de Seleção de Transformações (PST) consiste em encontrar uma sequência de transformações para um programa de entrada de forma a obter bom desempenho no código final.

Uma técnica para mitigar o problema citado é representar o conhecimento do programa de entrada, e projetar um formalismo que possa ser utilizado por um sistema automático de raciocínio para inferir uma solução aceitável para o PST.

Neste capítulo, é descrita uma tentativa de formalizar a representação do conhecimento com o objetivo de mitigar o PST.

As principais contribuições são:

- descrever representações do conhecimento para caracterizar programas;
- apresentar uma técnica para avaliar essas representações; e

- demonstrar a utilização de uma boa representação.

Os resultados indicam que uma boa representação do conhecimento é capaz de chegar a 81% de proximidade do melhor resultado possível de uma base de conhecimento. Além disso, quando utilizada por um sistema de geração de código, pode obter, para diversos casos, um código alvo melhor do que os alcançados por outras estratégias.

3.1 Caracterização de Programas

Programas de computador podem ser representados por características dinâmicas ou estáticas, que auxiliam em parametrizar o sistema de geração de código. As características dinâmicas descrevem o comportamento do programas no que diz respeito à sua execução. Por outro lado, características estáticas descrevem as estruturas algorítmicas do programa.

O atrativo das características dinâmicas é que se considera características tanto do *hardware* quanto do programa. Porém, a desvantagem é a necessidade da execução do programa, além da dependência de plataforma.

Alternativamente, características estáticas são independentes de plataforma e não requerem a execução do programa. Entretanto, essas representações não consideram os dados de entrada, que são elementos que podem alterar o comportamento e consequentemente causar alteração de parâmetros do sistema de geração de código.

Entre as representações de programas apresentadas na literatura, este trabalho avalia as seguintes:

- Dinâmica
 1. **Performance Counters (PC)**: são características resultantes da execução do programa e consistem nos contadores de desempenho do *hardware* disponíveis. Diversos trabalhos utilizaram PC como esquema de representação de programas (Cavazos et al., 2007; Lima et al., 2013; Queiroz Junior e da Silva, 2015). A Tabela 3.1 apresenta PC utilizados nesta dissertação.
- Estáticas
 1. **Dados de Compilação (DC)**: São características que descrevem os relacionamentos entre as entidades do programa, definidas pela representação intermediária utilizada pelo sistema de geração de código, assim como pela respectiva arquitetura de *hardware*. O uso dessas características foi proposto

Tabela 3.1: *Performance Counters* utilizados neste trabalho.

Classe	Características				
Cache	PAPILL2.DCR	PAPILL2.TCR	PAPILL3.ICR	PAPILL3.TCM	PAPILL1.LDM
	PAPILL2.TCA	PAPILL2.DCA	PAPILL3.ICA	PAPILL3.TCR	PAPILL2.ICA
	PAPILL2.DCW	PAPILL2.TCW	PAPILL3.DCR	PAPILL3.DCA	PAPILL2.ICM
	PAPILL1.ICM	PAPILL2.DCH	PAPILL3.TCA	PAPILL3.TCW	PAPILL2.ICH
	PAPILL1.DCM	PAPILL1.TCM	PAPILL3.DCW	PAPILL2.ICR	
	PAPILL2.TCM	PAPILL2.DCM	PAPILL2.STM	PAPILL1.STM	
Ramificação	PAPILBR_PRC	PAPILBR_MSP	PAPILBR_CN	PAPILBR_TKN	
	PAPILBR_NTK	PAPILBR_UCN	PAPILBR_INS		
Ponto Flutuante	PAPIVVEC_SP	PAPIVVEC_DP			
	PAPIFDV_INS	PAPIFP_OPS	PAPIDP_OPS	PAPIFP_INS	PAPISP_OPS
TLB	PAPITLB_DM	PAPITLB_IM			
Ciclos	PAPIREF_CYC	PAPITOT_CYC	PAPISTL_ICY	PAPISTL_ICY	
Instruções	PAPITOT_INS				

por Queiroz Junior e da Silva (2015), e sua utilização é limitada pelos dados fornecidos pelo compilador, mesmo que exista uma relação direta com o código-fonte. Tais características são apresentadas na Tabela 3.2.

Tabela 3.2: Dados de Compilação utilizados neste trabalho.

Classe	Característica
Instruções Binárias	Número de instruções Add
	Número de instruções Sub
Instruções de Memória	Número de instruções Store
	Número de instruções Load
	Número de instruções de memória
	Número de instruções GetElementPtr
	Número de instruções Alloca
Instruções de Terminação	Número de instruções Ret
	Número de instruções Br
Outras Instruções	Número de instruções ICmp
	Número de instruções PHI
	Número de instruções de máquina impressas
	Número de instruções Call
Funções	Número de funções não-externas
Blocos Básicos	Número de blocos básicos
Instruções de Ponto Flutuante	Número de instruções de ponto flutuante
Instruções Totais	Número de instruções (de todos os tipos)

2. **Características Numéricas (CN):** são características extraídas das relações entre as entidades do programa, que são definidas pelas especificidades das linguagens de programação. Elas foram propostas no trabalho de Namolaru et al. (2010), no qual foram produzidas sistematicamente por experimentos. Namolaru et al. (2010) ainda provaram suas influências na parametrização de sistemas de geração de código. Essas características são apresentadas na Tabela 3.3.

Além dessas representações, esta dissertação utiliza também uma representação simbólica, similar a um DNA, na qual cada instrução da linguagem intermediária, utilizada pelo sistema de geração de código, é representada por um gene. Essa representação é uma

Tabela 3.3: Características Numéricas utilizadas neste trabalho.

Classe	Características
Blocos Básicos	Número de blocos básicos no método
	Número de blocos básicos com um único predecessor
	Número de blocos básicos com um único predecessor e um único sucessor
	Número de blocos básicos com um único predecessor e dois sucessores
	Número de blocos básicos com um único sucessor
	Número de blocos básicos com dois predecessores e um sucessor
	Número de blocos básicos com mais de dois predecessores
	Número de blocos básicos com mais de dois sucessores
	Número de blocos básicos com mais de dois sucessores e mais de dois predecessores
	Número de blocos básicos com número de instruções maior do que 500
	Número de blocos básicos com número de instruções no intervalo [15, 500]
	Número de blocos básicos com número de instruções menor do que 15
	Número de blocos básicos com dois predecessores
	Número de blocos básicos com dois sucessores
Número de blocos básicos com dois sucessores e dois predecessores	
Operações Binárias	Número de operações binárias bit a bit no método
	Número de operações binárias de ponto flutuante no método
	Número de operações binárias de inteiros no método
Instruções de Chamada	Número de chamadas que retornam um ponto flutuante
	Número de chamadas que retornam um ponteiro
	Número de chamadas que retornam um inteiro
	Número de chamadas com ponteiros como argumentos
	Número de chamadas com número de argumentos maior do que 4
	Número de chamadas diretas no método
Grafo de Fluxo de Controle	Número de chamadas indiretas (e.g. via ponteiros) no método
	Número de arestas críticas no grafo de fluxo de controle
	Número de ramificações condicionais no método
	Número de arestas no grafo de fluxo de controle
	Número de ramificações incondicionais no método
Instruções de Conversão	Número de nós phi nos blocos básicos
	Número de instruções de conversão de ponto flutuante
Funções	Número de instruções de conversão de inteiros
	Número de funções
Instruções de Memória	Número de instruções <i>store</i>
	Número de instruções <i>load</i>
	Número de instruções de endereçamento de memória
Outras instruções	Número de instruções de vetores
	Número de instruções <i>getElementPtr</i>
	Número de instruções no método
	Número de instruções <i>switch</i> no método
	Número de instruções de término
	Número de instruções agregadas
Número de instruções de associação no método	

extensão da proposta por Sanches e Cardoso (2010). A vantagem da utilização do DNA como representação de código é que ela captura todas as estruturas do programa enquanto codifica todas as suas instruções. A representação simbólica proposta é apresentada na Tabela 3.4.

Características PC são extraídas com ferramentas que analisam a execução do programa; enquanto DC, CN e DNA são extraídas pelo sistema de geração de código.

Tabela 3.4: Codificação do DNA.

Regras de transformação							
Br	A	FSub	P	Load	e	AddrSpaceCast	t
Switch	B	FMul	Q	Store	f	FPTrunc	u
IndirectBr	C	FDiv	R	Alloca	g	FPExt	v
Ret	D	FRem	S	Fence	h	FPToUI	x
Invoke	E	Shl	T	AtomicRMW	i	FPToSI	w
Resume	F	LShr	U	AtomicCmpXchg	j	ICmp	y
Unreachable	G	AShr	V	GetElementPtr	k	FCmp	x
Add	H	And	X	Trunc	l	Select	0
Sub	I	Or	W	ZExt	m	VAArg	1
Mul	J	Xor	Y	SExt	n	LandingPad	2
UDiv	K	ExtractElement	Z	UIToFP	o	PHI	3
SDiv	L	InsertElement	a	SIToFP	p	Call	4
URem	M	ShuffleVector	b	PtrToInt	q	others	5
SRem	N	ExtractValue	c	IntToPtr	r		
FAdd	O	InsertValue	d	BitCast	s		

3.2 Sistema de Geração de Código

O objetivo de um sistema de geração de código é produzir código alvo para uma arquitetura específica a partir de um código fonte (Aho et al., 2006). Esse processo, que é dividido em diversas etapas, aplica transformações ao código fonte com o objetivo de melhorar a qualidade do código alvo. Entretanto, encontrar uma boa sequência de transformações para um programa em particular é uma tarefa complexa, especialmente por causa do tamanho do espaço de busca.

Um caminho para mitigar esse problema é extrair conhecimento do sistema de geração de código e construir um formalismo capaz de suportar a seleção da sequência de transformações para um respectivo programa. Uma técnica simples para extrair conhecimento é aplicar uma fase de treinamento para o sistema, na qual vários programas são compilados com diferentes sequências de transformações. Depois desse processo, é possível extrair e relacionar boas sequências e seus respectivos programas.

Baseado nesse formalismo (boas sequências de transformações e seus respectivos programas), implementar um sistema de geração de código eficiente transforma-se em um problema de identificar programas similares.

3.3 Reações

Esta dissertação propõe a utilização de *reações* para identificar a similaridade entre dois programas.

Hipótese. *Dois ou mais programas são similares se reagem da mesma forma quando se aplicam a eles as mesmas sequências de transformações.*

Validação. *É possível obter curvas de desempenho com o mesmo comportamento, para os programas P_x e P_y , aplicando as mesmas sequências de transformações. Isso indica que ambos os programas reagem identicamente, tendo um alto grau de similaridade. Um método simples para verificar a hipótese é (1) compilar os programas com as mesmas sequências; (2) plotar o gráfico de desempenho para ambos os programas; e (3) comparar o comportamento de cada curva. Como apresentado na Figura 3.1, os programas `adpcm_c` e `n-body` são similares, pois possuem um comportamento parecido em relação às reações, diferente do programa `ackermann`, que reage de forma diferente. Esse padrão ocorre para programas que são ou não similares.*

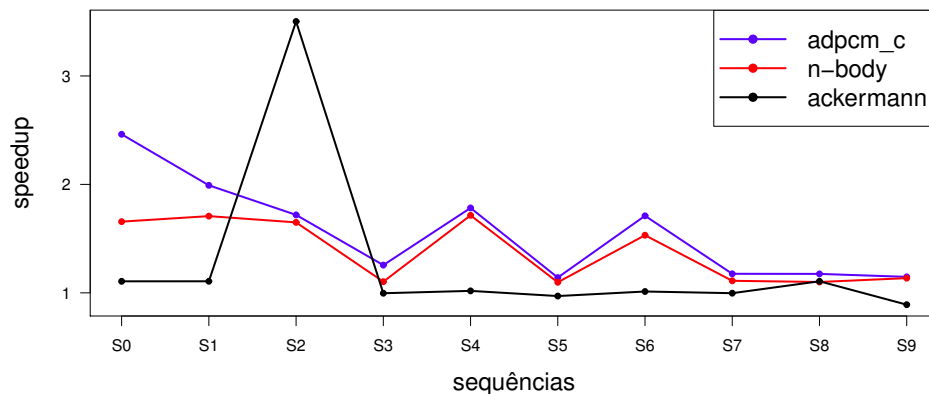


Figura 3.1: Desempenho sobre uma compilação sem a utilização de transformações.

Naturalmente, é necessário uma metodologia para identificar programas similares baseada nas suas características. Assim, baseado na premissa de que *reações* é uma boa estratégia para identificar similaridades, existe a necessidade de especificar o coeficiente de similaridade que, dadas as características de dois programas, determine se estes reagem similarmente.

3.3.1 Coeficientes para Identificar Reações Similares

Esta dissertação utiliza os coeficientes aplicados nos trabalhos de Lima et al. (2013) e Queiroz Junior e da Silva (2015) para identificar programas similares. Esses coeficientes são os seguintes:

Cosseno (CO) Neste coeficiente, a similaridade entre os programas P_x e P_y é obtida por:

$$sim(P_x, P_y) = \frac{\sum_{w=1}^M (P_{xw} \times P_{yw})}{\sqrt{\sum_{w=1}^M (P_{xw})^2} \times \sqrt{\sum_{w=1}^M (P_{yw})^2}}$$

Euclidiano (EU) Neste coeficiente, a similaridade entre os programas P_x e P_y é obtida por:

$$sim(P_x, P_y) = \frac{1}{\sqrt{\sum_{w=1}^M (P_{xw} - P_{yw})^2}}$$

Jaccard (JA) Neste coeficiente, a similaridade entre os programas P_x e P_y é obtida por:

$$sim(P_x, P_y) = \frac{1}{M} \sum_{w=1}^M \frac{\min(P_{xw}, P_{yw})}{\max(P_{xw}, P_{yw})}$$

Em cada coeficiente, M é o número de características.

Além desses 3 coeficientes, neste trabalho são utilizados:

Support Vector Machine (SVM) Trata-se de um modelo de aprendizado supervisionada, que analisa os dados utilizados para classificação (Scholkopf, 2002).

Needleman-Wunsch (NW) Trata-se de um algoritmo (Needleman e Wunsch, 1970) que servirá para comparar programas baseado na representação por DNA. É amplamente utilizado na literatura para comparação de DNAs biológicos. Dessa forma, para determinar uma reação similar entre dois programas, é avaliada a pontuação entre o alinhamento de seus DNAs.

3.3.2 Metodologia para Encontrar o Melhor Coeficiente

O coeficiente de similaridade que é capaz de identificar se duas curvas de *speedup* são similares, baseado no seu comportamento e amplitude, é considerado o melhor. O comportamento refere-se ao ganho de desempenho, se houve *speedup* ou *slowdown*. A amplitude refere-se a quanto houve de ganho ou perda de desempenho.

É possível descrever o comportamento do programa P_x , como mostrado na Tabela 3.5, baseado na análise de geração de código das sequências de transformações de 0 até n .

Tabela 3.5: Comportamento de P_x , quando compilado com as sequências de S_0 à S_n .

	S_0	S_1	S_2	...	S_n
S_0	1	T_0/T_1	T_0/T_2	...	T_0/T_n
S_1	T_1/T_0	1	T_1/T_2	...	T_1/T_n
S_2	T_2/T_0	T_2/T_1	1	...	T_2/T_n
...	1	...
S_n	T_n/T_0	T_n/T_1	T_n/T_2	...	1

Na Tabela 3.5, a linha i representa o ganho de desempenho provido pela aplicação das sequências, no qual S_i é o *baseline*, e T_i/T_j é o *speedup* ou *slowdown*.

Existe a possibilidade de verificar se houve ganho ou perda de desempenho para cada entrada ij . Isso pode ser visto nas tabelas correspondentes de P_x e P_y . A metodologia consiste em utilizar a parte superior à diagonal para se medir o comportamento. Assim para cada par (s_x, s_y) de desempenho comparado, a função $Coeff(s_x, s_y)$ mede a semelhança entre os comportamentos e amplitudes dos *speedups* referentes a s_x e s_y , como o seguinte:

$$Coeff(s_x, s_y) = \begin{cases} \frac{\min(s_x, s_y)}{\max(s_x, s_y)}, & \text{se } \neg(s_x > 1 \oplus s_y > 1) = \text{verdadeiro} \\ 0, & \text{caso contrário} \end{cases}$$

Considerando n sequências de transformações, o melhor coeficiente é o que obtém o maior valor de $MCoeff$, que é dado por:

$$MCoeff = \sum_{i=0}^n \sum_{j=i+1}^n Coeff(s_{ij}, s'_{ij})$$

no qual s_{ij} e s'_{ij} são os *speedups* obtidos por P_x e P_y , respectivamente.

3.4 Base de Sequências de Transformações

Com o intuito de avaliar *reações* entre dois programas e também o desempenho da representação do conhecimento para caracterização do programa, é necessário criar uma base de sequências de transformações. Esse processo é realizado da seguinte maneira:

Sistema de geração de código A infraestrutura de compilação LLVM 3.7.1 (Lattner, 2017).

Transformações A criação das sequências avalia 135 transformações disponíveis na LLVM 3.7.1 mostradas na Tabela 3.6.

-aa-eval -adce -add-discriminators -alignment-from-assumptions -alloca-hoisting -always-inline -argpromotion -assumption-cache-tracker -atomic-expand -barrier -basicaa -basiccg -bb-vectorize -bdce -block-freq -bounds-checking -branch-prob -break-crit-edges -cfl-aa -codegenprepare -consthoist -constmerge -constprop -correlated-propagation -cost-model -count-aa -da -dce -deadargelim -deadarghaX0r -delinearize -die -divergence -domfrontier -domtree -dse -dwarfefehprepare -early-cse -elim-avail-extern -flattencfg -float2int -functionattrs -generic-to-nvvm -globaldce -globalopt -globalsmodref-aa -gvn -indvars -inline -inline-cost -instcombine -instcount -instnamer -instsimplify -intervals -ipconstprop -ipsccp -irce -iv-users -jump-threading -lazy-value-info -lcssa -libcall-aa -licm -lint -load-combine -loop-accesses -loop-deletion -loop-distribute -loop-extract -loop-extract-single -loop-idiom -loop-instsimplify -loop-interchange -loop-reduce -loop-reroll -loop-rotate -loop-simplify -loop-unroll -loop-unswitch -loop-vectorize -loops -lower-expect -loweratomic -lowerbitsets -lowerinvoke -lowerswitch -mem2reg -mempyopt -memdep -mergfunc -mergereturn -mldst-motion -module-debuginfo -nary-reassociate -no-aa -partial-inliner -partially-inline-libcalls -place-backedge-safepoints-impl -place-safepoints -postdomtree -prune-eh -reassociate -reg2mem -regions -rewrite-statepoints-for-gc -rewrite-symbols -safe-stack -sancov -scalar-evolution -scalarizer -scallarrepl -scallarrepl-ssa -sccp -scev-aa -scoped-noalias -separate-const-offset-from-gep -simplifycfg -sink -sjlfehprepare -slp-vectorizer -slsr -speculative-execution -sroa -strip -strip-dead-debug-info -strip-dead-prototypes -strip-debug-declare -strip-nondebug -structurizecfg -tailcallelim -targetlibinfo -tbaa -tti -verify
--

Tabela 3.6: Transformações da infraestrutura LLVM 3.7.1 utilizadas.

Programas treino São compostos por programas retirados da suíte de testes da LLVM (Lattner, 2017), e do *The Computer Language Benchmarks Game* (Gouy, 2017). Esses programas foram utilizados no trabalho de Purini e Jain (2013). A Tabela 3.7 apresenta os programas treino.

Reduzindo o Espaço de Busca Este trabalho utiliza um processo com cinco passos:

1. Reduz o espaço de busca utilizando um Algoritmo Genético (AG);
2. Extrai as melhores sequências de transformações para cada programa treino;
3. Adiciona a essas sequências de transformações as 10 boas sequências encontradas por Purini e Jain (2013), bem como as 3 sequências dos níveis de compilação da LLVM (-01, -02 e -03);

Tabela 3.7: Programas treino.

Suíte de testes da LLVM			
ackermann (T00)	flops-3 (T14)	mandel (T28)	queens-mcgill (T42)
ary3 (T01)	flops-4 (T15)	mandel-2 (T29)	quicksort (T43)
bubblesort (T02)	flops-5 (T16)	matrix (T30)	random (T44)
chomp (T03)	flops-6 (T17)	methcall (T31)	realmm (T45)
dry (T04)	flops-7 (T18)	misr (T32)	recursive (T46)
dt (T05)	flops-8 (T19)	n-body (T33)	reedsolomon (T47)
fannkuch (T06)	fp-convert (T20)	nsieve-bits (T34)	richards_benchmark (T48)
fbench (T07)	hash (T21)	ourafft (T35)	salsa20 (T49)
fbench (T08)	heapsort (T22)	oscar (T36)	sieve (T50)
fib2 (T09)	himenobmtxpa (T23)	partialsums (T37)	spectral-norm (T51)
fdry (T10)	huffbench (T24)	perlin (T38)	strcat (T52)
flops (T11)	intmm (T25)	perm (T39)	towers (T53)
flops-1 (T12)	lists (T26)	pi (T40)	treесort (T54)
flops-2 (T13)	lpbench (T27)	queens (T41)	whetstone (T55)
<i>The Computer Language Benchmarks Game</i>			
binary-trees (T56)	fasta-redux (T58)	pidigits (T60)	
fasta (T57)	mandelbrot (T59)	regex-dna (T61)	

4. Avalia cada programa treino utilizando as 62+13 sequências; e
5. Grava na base de dados para cada programa a tupla \langle programa treino, boas sequências de transformações ¹ \rangle .

O AG consiste em gerar aleatoriamente uma população inicial, a qual será evoluída em um processo iterativo. Esse processo envolve a escolha dos pais; aplicação de operadores genéticos; evoluir novos indivíduos; e finalmente o elitismo, que decidirá quais indivíduos irão compor a nova geração. Esse processo iterativo é realizado até que um critério de parada for satisfeito.

A primeira geração é composta por indivíduos que são produzidos por uma amostra uniforme do espaço de transformação. Evoluir uma população inclui a aplicação de dois operadores genéticos: *crossover* e mutação. O primeiro operador possui uma probabilidade de 60% de criar um novo indivíduo. Nesse caso, uma estratégia de torneio ($Tour = 5$) seleciona seus geradores. O segundo operador, mutação, possui uma probabilidade de 40% de modificar um indivíduo. Além disso, cada indivíduo possui um tamanho inicial arbitrário, que pode variar de 1 até $|Espaço\ de\ Transformação|$. Desse modo, o operador de *crossover* pode ser aplicado a indivíduos de diferentes tamanhos. Nesse caso, o tamanho do novo indivíduo é a média do tamanho de seus pais. Quatro tipos de mutação são utilizados:

1. Inserir uma nova transformação em um ponto aleatório;
2. Remover uma transformação de um ponto aleatório;

¹Uma boa sequência de transformações é aquela que provém para o programa um tempo de execução menor do que o nível de transformações da LLVM que obteve melhor desempenho.

3. Trocar duas transformações de pontos aleatórios; e
4. Alterar uma transformação em um ponto aleatório.

Todos os operadores possuem a mesma probabilidade de ocorrência, além de somente uma mutação ser aplicada ao indivíduo selecionado para ser transformado. Esse processo iterativo utiliza elitismo, o que mantém o melhor indivíduo na próxima geração. Além disso, executa pelo menos 100 gerações com 50 indivíduos, e finaliza se o desvio padrão da aptidão atual for menor do que 0,01, ou se a melhor aptidão não for alterada em três gerações consecutivas.

A estratégia utilizada para reduzir o espaço de busca é similar à estratégia proposta por Purini e Jain (2013) e Martins et al. (2016).

3.5 Avaliando Representações de Conhecimento

As seções seguintes descrevem as avaliações realizadas com o intuito de determinar os melhores coeficiente e estratégia para caracterizar programas.

3.5.1 Ambiente Experimental

Arquitetura Intel(R) Core(TM) i7-3770 CPU 3.4GHz com 8GB RAM executando sistema operacional Ubuntu 14.04 x64 com kernel 4.2.0-41.

Compilador A infraestrutura de compilação LLVM 3.7.1² (Lattner e Adve, 2004).

Extração de características PC são extraídos com a ferramenta PAPI (Mucci e Mohan, 2017). DC são características disponibilizadas pela infraestrutura LLVM. Dois módulos extratores foram implementados com o intuito de extrair CN e DNA durante o processo de compilação, da representação intermediária da LLVM.

Representando programas Este trabalho examina duas diferentes abordagens para representar programas: (1) funções quentes (QUT); e (2) programas completos (CPT). Funções quentes são as que possuem o maior custo do programa, o que significa que consomem a maior parte do tempo de execução. O coeficiente de similaridade examina programas baseado nas suas funções quentes e determina se são similares ou não. O algoritmo proposto por Wu e Larus (1994) foi utilizado para identificar a função quente de cada programa.

²<http://www.llvm.org>

SVM A biblioteca SKLEARN (Pedregosa et al., 2011) foi utilizada para calcular este coeficiente.

Programas A fase de testes utilizou programas que pertencem ao *Collective Benchmark* (CBENCH) (Fursin, 2017) e ao *Polyhedral Benchmark Suite* (Polybench) (Pouchet, 2017). Para os programas do cBench utilizou-se a entrada 1 e para o Polybench a entrada large.

Sequências de Transformações Este experimento utiliza 75 sequências para avaliar *reações*: 62 encontradas pelo AG durante o processo de redução do espaço de busca; as 10 sequências encontradas por Purini e Jain (2013); e 3 sequências providas pela LLVM (-01, -02, e -03).

Tempo de Execução Cada programa foi executado 100 vezes para assegurar resultados precisos. Além disso, foram descartados 20% dos resultados: os 10% melhores e os 10% piores. Assim, a média aritmética do tempo de execução é calculada com base em 80% dos dados.

3.5.2 Os Melhores Resultados

Para cada programa teste, a Tabela 3.8 apresenta seu programa mais similar, no que diz respeito aos programas treino, e o valor máximo de **MCoeff** alcançado (**bMCoeff**).

Tabela 3.8: bMCoeff

Programas cBench								
programa	similar	valor	programa	similar	valor	programa	similar	valor
adpcm_c (C00)	T43	973,81	gsm (C10)	T36	633,14	rijndael_e (C20)	T07	552,13
adpcm_d (C01)	T41	1016,72	jpeg_c (C11)	T32	637,31	rsynth (C21)	T48	1158,37
bitcount (C02)	T41	1082,03	jpeg_d (C12)	T42	595,48	sha (C22)	T06	1122,57
blowfish_d (C03)	T33	1087,25	lame (C13)	T27	814,21	susan_c (C23)	T33	906,77
blowfish_e (C04)	T33	1090,80	mad (C14)	T39	946,59	susan_e (C24)	T06	927,34
bzip2d (C05)	T56	919,08	patricia (C15)	T03	1175,13	susan_s (C25)	T50	1034,76
bzip2e (C06)	T48	891,61	pgp_d (C16)	T32	981,34	tiff2bw (C26)	T42	876,43
CRC32 (C07)	T21	1118,84	pgp_e (C17)	T06	911,81	tiffrgba (C27)	T42	874,93
dijkstra (C08)	T03	1188,44	qsort1 (C18)	T08	1137,78	tiffdither (C28)	T42	903,93
ghostscript (C09)	T22	419,72	rijndael_d (C19)	T59	660,21	tiffmedian (C29)	T43	849,50
Programas Polybench								
programa	similar	valor	programa	similar	valor	programa	similar	valor
2mm (P00)	T25	1352,41	fdtd-2d (P10)	T07	1193,10	mvt (P20)	T57	1109,52
3mm (P01)	T25	1380,98	floyd-warshall (P11)	T25	1106,82	nussinov (P21)	T57	1066,27
adi (P02)	T07	1159,72	gemm (P12)	T20	1089,59	seidel-2d (P22)	T07	1125,84
2mm (P03)	T57	1089,14	gemver (P13)	T57	1057,15	symm (P23)	T15	1204,63
big (P04)	T29	1036,28	gesummv (P14)	T25	985,16	syr2k (P24)	T15	1111,42
cholesky (P05)	T45	1102,08	gramschmidt (P15)	T28	913,13	syrk (P25)	T14	1227,14
correlation (P06)	T07	1203,89	heat-3d (P16)	T57	1192,77	trisolv (P26)	T57	1030,43
covariance (P07)	T57	1125,70	jacobi-2d (P17)	T33	1083,28	trmm (P27)	T57	1205,69
deriche (P08)	T45	1131,50	lu (P18)	T45	1156,71			
doitgen (P09)	T47	1338,01	ludcmp (P19)	T45	1071,26			

O maior valor possível para **bMCoeff** é 2775 porque se consideram 75 sequências para o processo de avaliação. Como apresentado na Tabela 3.8, esse valor não é atingido. Assim,

é importante considerar que tal métrica tenta modelar o comportamento juntamente com a amplitude alcançados pelos desempenhos obtidos. Assim, mesmo que o maior valor não seja alcançado, é importante identificar o melhor valor possível, e igualmente qual coeficiente de similaridade e caracterização produzirá o valor mais próximo ao melhor alcançável.

3.5.3 Resultados e Discussão

A Tabela 3.9 apresenta os resultados obtidos pelas estratégias avaliadas, os quais referem-se à distância para o melhor valor possível ($\frac{MCoeff}{bMCoeff}$).

O melhor valor médio foi obtido por CPT-CN com CO. Outras estratégias perderam por até 10,78% de desempenho.

Vale a pena ressaltar o desempenho inesperado para PC, o qual foi a única característica dinâmica avaliada. De fato, PC obteve a menor média entre todas as estratégias. O melhor valor foi até 4,51% pior do que CPT-CN.

CN apresentou resultados consistentes considerando todos os coeficientes, tendo a menor variância; $0,58 \times 10^{-4}$, e $1,70 \times 10^{-4}$, para QUT e CPT respectivamente. Alternativamente, outras estratégias obtiveram variâncias de $6,12 \times 10^{-4}$, e $2,77 \times 10^{-4}$, para PC e DC respectivamente.

A maior variância entre as médias foi obtida por PC. Isso significa que apesar de essa representação obter bons desempenhos com o coeficiente JA, os resultados com outros coeficientes foram discrepantes, chegando a $\simeq 6,57\%$ de diferença entre SVM e JA.

DNA obteve uma média 1,51% menor do que a melhor estratégia. Porém, em termos globais, foi 30,77% pior quando comparada novamente à melhor estratégia. Entre as estratégias QUT, DNA e JA obtiveram resultados satisfatórios, o que indica a possibilidade de representar o programa focando em sua função quente.

CPT-CN foi o melhor esquema de representação de programas, por causa de quatro razões:

1. obteve o melhor MCoef;
2. possui estabilidade para alcançar resultados perfeitos quando o coeficiente de similaridade é alterado;
3. obteve uma das menores variâncias entre os melhores resultados; e
4. seus piores resultados não são tão baixos quanto os alcançados por outras representações.

Tabela 3.9: Resultados obtidos para cada programa de cBench e Polybench

	GUT					CPT											
	DNA	CN				DC				PC				CN			
		NW	CO	EU	JA	SVM	CO	EU	JA	SVM	CO	EU	JA	SVM	CO	EU	JA
C00	0,82	0,71	0,72	0,99	0,72	0,88	0,87	0,76	0,87	0,96	0,74	0,85	0,74	0,71	0,87	0,87	0,87
C01	0,94	0,77	0,67	0,88	0,67	0,88	0,90	0,86	0,90	1,00	0,76	0,87	0,76	0,77	0,90	0,90	0,90
C02	0,85	0,85	0,88	0,88	0,88	0,98	0,91	0,98	0,91	0,91	0,73	0,85	0,73	0,76	0,90	0,81	0,90
C03	0,85	0,75	0,77	0,77	0,77	0,92	0,77	0,71	0,77	0,69	0,69	0,75	0,69	0,69	0,77	0,77	0,77
C04	0,86	0,77	0,76	0,76	0,76	0,93	0,76	0,72	0,76	0,68	0,68	0,78	0,68	0,68	0,76	0,76	0,76
C05	0,80	0,82	0,75	0,95	0,75	0,82	0,75	0,61	0,75	0,87	0,87	0,90	0,87	0,96	0,75	0,75	0,75
C06	0,81	0,98	0,85	0,92	0,85	0,98	0,85	0,64	0,85	0,92	0,86	0,86	0,86	0,91	0,85	0,85	0,85
C07	0,70	0,68	0,58	0,70	0,58	0,52	0,80	0,54	0,80	0,73	0,73	0,74	0,73	0,89	0,78	0,93	0,78
C08	0,79	0,58	0,81	0,72	0,81	1,00	0,74	0,96	0,74	0,96	0,87	0,91	0,87	1,00	0,74	0,69	0,74
C09	0,99	0,73	0,79	0,70	0,79	0,77	0,77	0,58	0,77	0,83	0,89	0,83	0,89	1,00	0,77	0,77	0,77
C10	0,75	0,70	0,85	0,85	0,85	0,84	0,93	0,61	0,93	0,65	0,65	0,76	0,65	1,00	0,93	0,93	0,93
C11	0,65	1,00	0,71	0,71	0,71	0,83	0,75	0,61	0,75	0,73	0,63	0,81	0,61	0,83	0,75	0,75	0,75
C12	0,74	0,95	0,85	0,85	0,85	0,91	0,88	0,67	0,88	0,90	0,71	0,79	0,71	0,91	0,88	0,88	0,88
C13	0,96	0,88	0,93	0,93	0,93	0,88	0,83	0,64	0,83	0,89	1,00	1,00	1,00	0,88	0,83	0,83	0,83
C14	0,82	0,75	0,89	0,84	0,89	0,84	0,76	0,57	0,76	0,69	0,69	0,82	0,69	0,66	0,76	0,76	0,76
C15	0,73	0,92	0,89	0,73	0,89	0,66	1,00	0,93	1,00	0,85	0,85	0,85	0,85	0,89	1,00	1,00	1,00
C16	0,87	1,00	1,00	1,00	1,00	0,62	0,79	0,61	0,79	0,65	0,65	0,83	0,65	0,89	0,79	0,79	0,79
C17	1,00	0,91	0,91	0,91	0,91	0,63	0,90	0,67	0,90	0,73	0,89	0,76	0,89	0,85	0,90	0,90	0,90
C18	0,92	0,71	0,93	0,82	0,93	0,92	0,76	0,75	0,76	0,78	0,83	0,83	0,83	0,86	0,96	0,96	0,96
C19	0,65	0,78	0,68	0,68	0,68	0,82	0,66	0,88	0,66	0,82	0,79	0,88	0,79	0,82	0,66	0,66	0,66
C20	0,63	0,76	0,58	0,60	0,58	0,75	0,78	0,93	0,78	0,75	0,78	0,93	0,62	0,75	0,78	0,78	0,78
C21	0,72	0,79	0,94	0,94	0,94	0,90	0,63	0,50	0,63	0,60	0,60	0,66	0,60	0,95	0,63	0,63	0,63
C22	0,91	0,90	0,90	0,90	0,90	0,87	0,87	0,70	0,87	0,74	0,74	0,70	0,74	0,92	0,87	0,92	0,87
C23	0,99	0,90	0,81	0,90	0,81	0,90	0,99	0,91	0,99	0,72	0,80	0,88	0,80	0,99	0,99	0,99	0,99
C24	1,00	0,87	0,83	0,87	0,83	0,87	1,00	0,84	1,00	0,73	0,87	0,87	0,87	1,00	1,00	1,00	1,00
C25	0,86	0,79	0,65	0,83	0,65	0,79	0,86	0,71	0,86	0,62	0,62	0,62	0,62	0,86	0,86	0,86	0,86
C26	0,90	0,80	0,88	0,88	0,88	0,73	0,88	0,62	0,88	0,90	0,66	0,76	0,66	0,86	0,88	0,88	0,88
C27	0,86	0,79	0,88	0,88	0,88	0,75	0,88	0,59	0,88	0,86	0,77	0,79	0,77	0,82	0,88	0,88	0,88
C28	0,84	0,76	0,74	0,74	0,74	0,69	0,74	0,59	0,74	0,73	0,65	0,69	0,65	0,87	0,74	0,74	0,74
C29	0,72	0,80	0,73	0,66	0,73	0,80	0,87	0,61	0,87	0,91	0,72	0,70	0,72	0,74	0,87	0,87	0,87
\bar{X}_c	0,82	0,81	0,80	0,82	0,80	0,81	0,82	0,70	0,82	0,78	0,75	0,80	0,74	0,85	0,83	0,83	0,83
> _c	5	4	3	3	3	5	5	3	5	3	1	3	1	10	6	8	6
P00	0,66	0,66	0,64	0,64	0,64	0,48	0,69	0,80	0,69	0,63	0,63	0,58	0,63	0,48	0,69	0,69	0,69
P01	0,72	0,67	0,68	0,76	0,68	0,52	0,80	0,79	0,80	0,86	0,61	0,61	0,61	0,58	0,61	0,72	0,61
P02	0,67	0,67	0,71	0,70	0,71	0,67	0,67	0,75	0,67	0,75	0,60	0,81	0,60	0,67	0,67	0,67	0,67
P03	0,84	0,79	0,79	0,82	0,79	0,60	0,66	0,91	0,66	0,79	0,76	0,79	0,76	0,77	0,66	0,77	0,66
P04	0,81	0,85	0,83	0,81	0,83	0,88	0,73	0,91	0,73	0,67	0,87	0,84	0,87	0,92	0,76	0,94	0,76
P05	0,63	0,81	0,80	0,87	0,80	0,81	0,81	0,87	0,81	0,65	0,68	0,82	0,68	0,77	0,81	0,81	0,81
P06	0,75	0,86	0,86	0,82	0,86	0,86	0,86	0,72	0,86	0,49	0,68	0,68	0,68	0,86	0,90	0,86	0,90
P07	0,71	0,83	0,75	0,98	0,75	0,66	0,76	0,85	0,76	0,55	0,70	0,70	0,70	0,85	0,65	0,63	0,65
P08	0,96	0,89	0,89	0,89	0,89	0,62	0,96	0,84	0,96	0,83	0,71	0,85	0,71	0,79	0,84	0,96	0,84
P09	0,64	0,46	0,63	0,62	0,63	0,46	0,59	0,73	0,59	0,62	0,55	0,61	0,55	0,64	0,59	0,60	0,59
P10	0,86	0,85	0,72	1,00	0,72	0,85	0,94	0,64	0,94	0,86	0,66	0,84	0,66	0,85	0,94	0,91	0,94
P11	0,85	0,79	0,81	0,73	0,81	0,60	0,86	0,95	0,86	0,87	0,82	0,65	0,82	0,73	0,65	0,65	0,65
P12	0,90	0,53	0,65	0,74	0,65	0,53	0,72	0,85	0,72	0,69	0,72	0,63	0,72	0,53	0,72	0,72	0,72
P13	0,85	0,87	0,78	0,88	0,78	0,89	0,84	0,88	0,84	0,78	0,78	0,87	0,78	0,81	0,84	0,75	0,84
P14	0,98	0,82	0,84	0,75	0,84	0,86	0,74	0,98	0,74	0,89	0,73	0,84	0,73	0,90	0,70	0,70	0,70
P15	0,76	0,87	0,79	0,96	0,79	0,91	0,91	0,91	0,91	0,80	0,75	0,75	0,75	0,84	0,91	0,91	0,91
P16	0,84	0,84	0,82	0,69	0,82	0,84	0,84	0,96	0,84	0,87	0,70	0,69	0,70	0,84	0,84	0,89	0,84
P17	0,63	0,94	0,80	0,83	0,80	0,94	0,81	0,86	0,81	0,64	0,76	0,79	0,76	0,94	0,58	0,59	0,58
P18	0,87	0,69	0,76	0,60	0,76	0,76	0,60	0,79	0,60	0,66	0,63	0,79	0,63	0,76	0,79	0,79	0,79
P19	0,86	0,79	0,70	0,78	0,70	0,79	0,73	0,80	0,73	0,60	0,73	0,62	0,73	0,88	0,85	0,88	0,85
P20	0,86	0,86	0,80	0,76	0,80	0,79	0,75	0,88	0,75	0,59	0,73	0,73	0,73	0,85	0,75	0,87	0,75
P21	0,92	0,92	0,86	0,86	0,86	0,91	0,86	0,88	0,86	0,69	0,72	0,73	0,72	0,91	0,92	0,86	0,92
P22	0,65	0,63	0,72	0,65	0,72	0,85	0,86	0,92	0,86	0,84	0,56	0,87	0,56	0,85	0,81	0,86	0,81
P23	0,69	0,80	0,87	0,87	0,87	0,62	0,59	0,84	0,59	0,89	0,72	0,77	0,72	0,81	0,61	0,61	0,61
P24	0,64	0,64	0,79	0,60	0,79	0,70	0,61	0,78	0,61	0,87	0,70	0,66	0,70	0,70	0,61	0,60	0,61
P25	0,82	0,58	0,75	0,75	0,75	0,58	0,76	0,83	0,76	0,77	0,69	0,80	0,69	0,58	0,76	0,62	0,76
P26	0,69	0,88	0,75	0,69	0,75	0,88	0,80	0,92	0,80	0,76	0,91	0,88	0,91	0,92	0,80	0,80	0,80
P27	0,82	0,79	0,69	0,82	0,69	0,65	0,81	0,87	0,81	0,87	0,72	0,82	0,72	0,87	0,63	0,72	0,63
\bar{X}_p	0,77	0,76	0,76	0,77	0,76	0,72	0,76	0,84	0,76	0,73	0,70	0,75	0,70	0,77	0,74	0,76	0,74
> _p	5	2	0	4	0	2	1	12	1	4	0	1	0	4	2	3	2
Valores Gerais																	
\bar{X}	0,80	0,78	0,78	0,80	0,78	0,77	0,79	0,76	0,79	0,76	0,73	0,77	0,72	0,81	0,78	0,79	0,78
>	10	6	3	7	3	7	6	15	6	7	1	4	1	14	8	11	8

No que diz respeito aos coeficientes, CO foi o melhor por três razões:

1. obteve os melhores valores de MCoeff;
2. foi o coeficiente que apresentou, extensivamente, os melhores resultados; e
3. ficou entre os coeficientes que alcançaram maior número de programas com o melhor resultado.

3.6 Sistema Automático de Geração de Código

A fim de avaliar o desempenho da melhor estratégia para caracterizar e encontrar programas similares, esta seção descreve um Sistema Automático de Geração de Código (SAGC) capaz de inferir a melhor sequência de transformações, baseado em experiências anteriores, para os programas teste.

O sistema emprega um modelo tradicional de aprendizagem de máquina, que é composto por fases *offline* e *online*. A primeira cria uma base de dados contendo boas sequências de transformações para programas diferentes. A última prevê uma boa sequência para o programa teste e gera o código alvo.

3.6.1 Fase Offline

Em um modelo de aprendizagem de máquina, a fase *offline* consiste em uma fase treino.

Nesta dissertação, a fase *offline* consiste na construção da base descrita na Seção 3.4. Como resultado, os dados de treino pertencem à mesma base de dados que foi gerada para avaliar *reações*, além da representação do conhecimento CPT-CN que caracteriza cada programa treino.

3.6.2 Fase Online

A fase *online* consiste em uma estratégia de Raciocínio Baseado em Casos (RBC), a qual realiza as seguintes etapas:

1. **Recuperar experiências anteriores.** Esta etapa é realizada extraindo as características CN do programa teste e calculando o valor da similaridade para cada programa que pertence à base de dados com o coeficiente CO. Por fim N sequências do programa treino com a similaridade mais alta são recuperadas.
2. **Reutilizar experiências anteriores.** Esta etapa é realizada gerando código para o programa teste utilizando as sequências de transformações recuperadas no passo anterior.
3. **Rever o resultado, avaliando o sucesso da solução.** Esta etapa é realizada executando o código gerado e medindo seu tempo de execução.

3.6.3 Metodologia

O sistema descrito foi avaliado utilizando o ambiente descrito na Seção 3.5.1. A avaliação considera 1, 3, 5 e 10 experiências anteriores. Para avaliar a execução de cada código gerado, realizam-se 10 execuções e, para evitar flutuações do tempo de execução, são descartados 20% dos dados: 10% maiores e 10% menores. Após isso, calcula-se a média de tempo com base nos 80% tempos restantes.

Para avaliar a efetividade do **SAGC**, esta seção o compara com três técnicas:

1. **AG com seleção por torneio (AG50)**: é a técnica descrita na Seção 3.4.
2. **AG com seleção por torneio (AG10)**: é similar à técnica descrita na Seção 3.4, exceto por executar sobre 10 gerações e 20 indivíduos.
3. **10 boas sequências (Best10)**: é a técnica proposta por Purini e Jain (2013). Os autores encontraram 10 sequências que consideraram boas, que são capazes de cobrir diversas classes de programas. Assim, nessa técnica os programas não vistos são compilados com todas as sequências, e o melhor código alvo é retornado.

A avaliação utiliza cinco métricas para analisar os resultados:

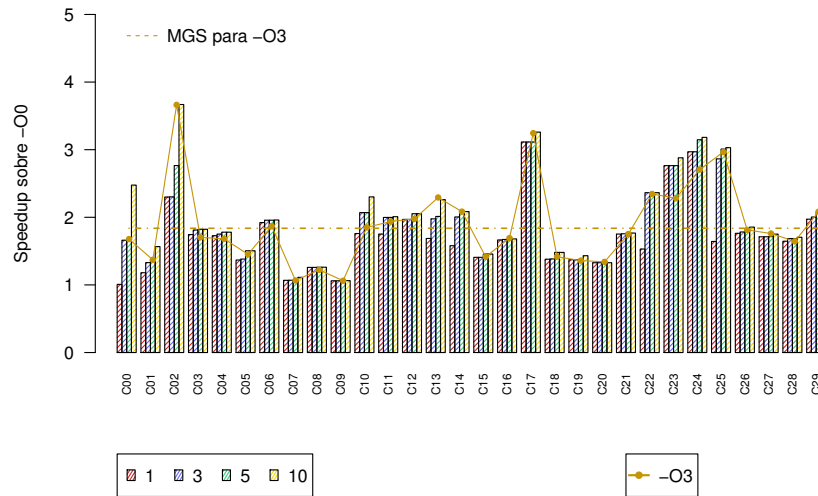
1. **MGS**: Média geométrica de *speedup*;
2. **NPS**: Número de programas com *speedup* maior do que o nível mais agressivo de transformação (-03);
3. **MLH**: Melhoria em relação ao nível -03;
4. **NS**: Número de sequências avaliadas; e
5. **TR**: O tempo de resposta da técnica.

O *speedup* é calculado como segue:

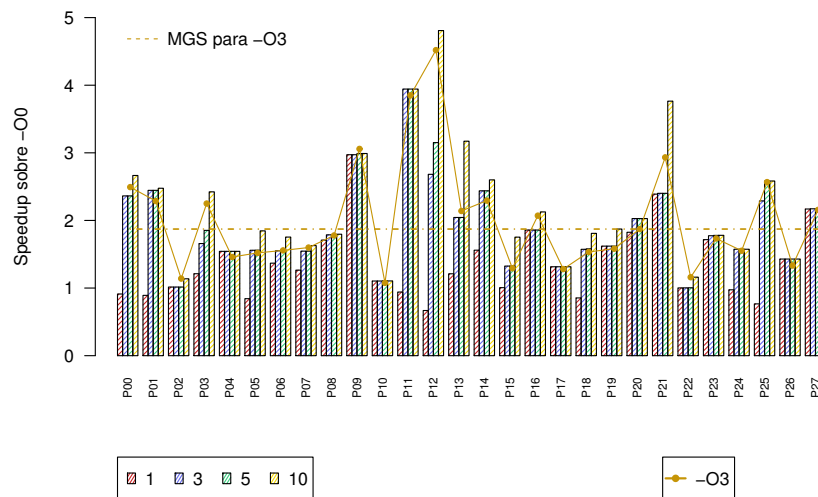
$$Speedup = \frac{Tempo_de_Execucao_Nivel_00}{Tempo_de_Execucao}$$

3.6.4 Visão Geral

Esta seção apresenta uma visão geral do desempenho do **SAGC**, utilizando **CN** como melhor esquema de representação, e **C0** como melhor coeficiente para identificar programas com reações similares. A Figura 3.2 mostra o *speedup* alcançado para cada programa teste.



(a) cBench



(b) Polybench

Figura 3.2: *Speedups* obtidos pelo SAGC proposto para os programas teste recuperando 1, 3, 5, e 10 casos anteriores.

MGS Os valores de MGS foram 1,456, 1,794, 1,829 e 1,976 para SAGC.1, SAGC.3, SAGC.5 e SAGC.10, respectivamente. SAGC.5 chegou próximo do desempenho do melhor nível -O3 do compilador (menos de 1%). Isso significa que considerando somente 5 casos anteriores o SAGC proposto é capaz de alcançar os tempos de execução do melhor

nível do compilador. Além disso, considerando 10 casos anteriores, **SAGC** aumenta o desempenho, chegando a 13,74% mais do que **-03**.

NPS SAGC.1 obteve uma cobertura de 25,86% dos programas avaliados, enquanto outras estratégias chegaram a 50,00%, 65,52% e 87,93% para **SAGC.3**, **SAGC.5** e **SAGC.10**. Isso indica que aumentando o número de experiências anteriores recuperadas, a cobertura cresce, diminuindo a quantidade de programas em que **SAGC.x** é incapaz de alcançar desempenho.

MLH Em relação à **-03**, a técnica obteve melhorias de 8,19%, 8,38%, 7,87% e 17,72% para **SAGC.1**, **SAGC.3**, **SAGC.5** e **SAGC.10**, respectivamente. A diferença fica maior quando se considera o declínio para os programas em que não houve melhoria em relação à **-03**: **SAGC.1** piorou em média 58,35%, enquanto em **SAGC.3** e **SAGC.5** houve declínio médio 22,27% e 22,61%, e em **SAGC.10**, somente 1,98%.

NS A questão importante relacionada a esta métrica é o quanto compensa avaliar mais sequências e aumentar o custo do sistema. De fato, é necessário encontrar um balanceamento entre esses dois valores. Analisando os resultados, é possível identificar que, de 1 para 10 experiências recuperadas, a **MGS** aumenta 33,78%, 3,53% e 14,67%, respectivamente, enquanto o **NPS** cresce 93,33%, 31,03% e 34,21%. Esses resultados indicam que aumentar o custo do sistema com mais avaliações com até 10 experiências recuperadas é uma boa escolha, pois gera melhores resultados.

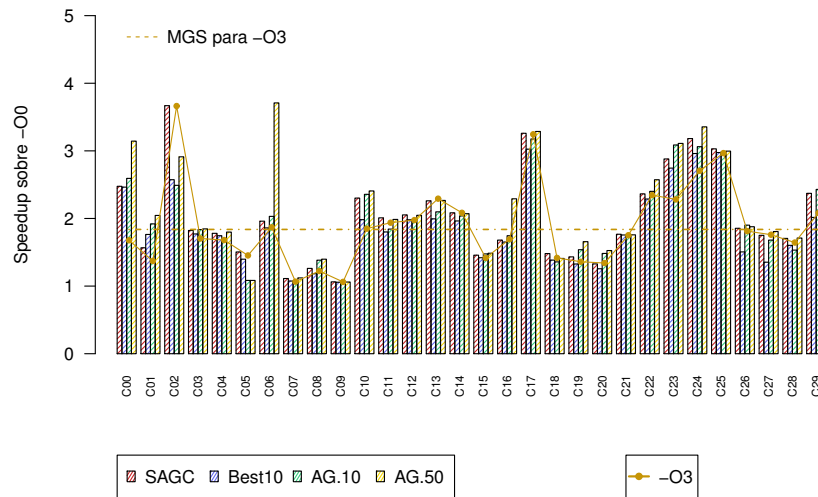
TR O valor correspondente a esta métrica é 41,3s (segundos), 105,3s, 168,9s e 320,8 para **SAGC.1**, **SAGC.3**, **SAGC.5** e **SAGC.10**, respectivamente. Esses tempos são proporcionais à métrica **NS**. A suposição sobre **TR** é a mesma sobre **NS**.

Para o conjunto de programas **cBench**, o desempenho obtido foi de 1,649, 1,788, 1,831 e 1,919 para 1, 3, 5 e 10 casos recuperados, respectivamente. A cobertura para esses programas foi de 30,00%, 43,33%, 66,67% e 83,33% também para os valores de **N** iguais 1, 3, 5 e 10, respectivamente.

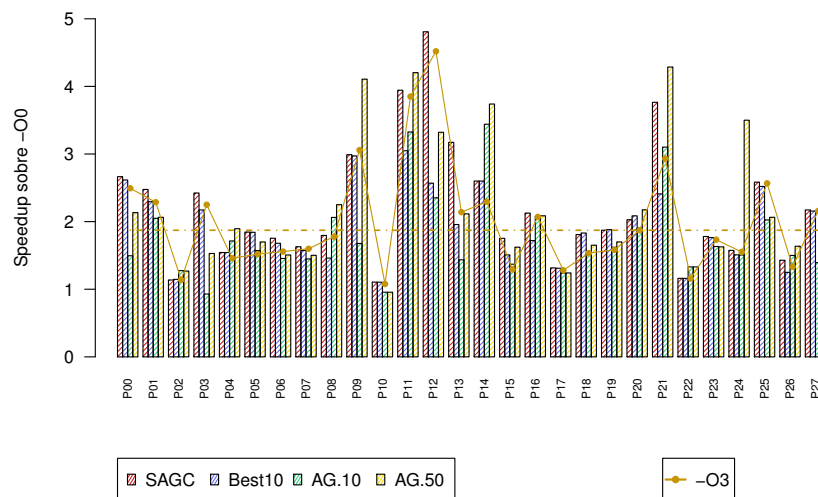
Se considerado o desempenho somente para os programas **Polybench**, os valores são mais altos recuperando um maior número de casos. Os valores de **MGS** foram 1,274, 1,799, 1,826 e 2,038 para os valores de **N** iguais a 1, 3, 5 e 10, respectivamente. A cobertura para esses programas também foi maior do que a média geral, chegando a 21,43%, 57,14%, 64,29% e 92,86% para 1, 3, 5 e 10 casos recuperados, respectivamente.

3.6.5 Comparação

Esta seção compara o SAGC proposto com outras estratégias. A Figura 3.3 apresenta os *speedups* para SAGC, Best10, AG10 e AG50.



(a) cBench



(b) Polybench

Figura 3.3: *Speedups* alcançados comparados com outras técnicas.

MGS A MGS alcançada por SAGC foi 1,919, superada somente por AG50, que alcançou 2,015.

Os valores para Best10 e AG10 foram 1,812 e 1,779, respectivamente. É importante

considerar que **SAGC** e **AG** possuem diferentes premissas. O primeiro consiste em um paradigma de aprendizagem de máquina, que retorna uma solução em poucas etapas, enquanto **AG** é um processo de compilação iterativa, que avalia diversas sequências para o programa. **SAGC** alcançou desempenho similar a **AG50**, com um diferença de somente 3,95%, em um tempo consideravelmente menor (99%). Além disso, **SAGC** superou as outras estratégias avaliadas.

NPS **SAGC** não alcançou os altos *speedups* atingidos por **AG50**, porém o valor para **NPS** foi maior, cobrindo 87,93% dos programas enquanto **AG50** cobre 68,97%. Isso significa que **AG50** obteve valores altos discrepantes em programas isolados, enquanto **SAGC** conseguiu bom desempenho para mais programas. **Best10** e **AG10** obtiveram 46,55% e 55,17% de cobertura, respectivamente.

MLH Em relação ao nível -03, **SAGC** obteve melhoria média de 17,72% nos programas em que alcançou cobertura, enquanto **Best10**, **AG10** e **AG50** obtiveram 15,82%, 22,15% e 43,20%, também em média. O alto número obtido por **AG50** é ofuscado quando se consideram os programas não cobertos por essa técnica, nos quais houve declínio médio de 27,88%, enquanto em **Best10** os programas decaíram em média 25,11% e em **AG10** 43,93%. Em decorrência da maior cobertura, **SAGC** possui uma média baixa de declínio em relação à -03, somente 1,98%.

NS Como técnicas de compilação iterativa, **AG10** e **AG50** avaliam um alto número de sequências para encontrar um resultado, tendo, em média, **NS** de 57,1 e 259,3, respectivamente. A estratégia **Best10** avalia o mesmo número de sequências que **SAGC**. 10 (10 sequências). Porém, **Best10** avalia sempre as mesmas 10 sequências, enquanto **SAGC** recupera uma experiência anterior específica baseada nas características do programa.

TR **AG50** foi a estratégia que mais consumiu tempo, levando mais de 69.018,0s para fornecer um resultado, em média. **AG10** gastou em média 13.693,8s para cada programa de entrada, enquanto **Best10** forneceu uma resposta depois de 390,7s, também em média. **SAGC** foi 17,89% mais rápida do que a estratégia **Best10**, avaliando o mesmo número de sequências em um tempo médio de 320,8s. Teoricamente, o tempo de resposta é essencialmente proporcional ao número de sequências avaliadas, porém **Best10** avalia as mesmas 10 sequências para cada programa teste, sequências que podem consumir mais tempo do que as boas sequências recuperadas por **SAGC**.

Para os programas **cBench**, as estratégias **Best10**, **AG10** e **AG50** obtiveram os valores de **MGS** iguais a 1,784, 1,882 e 2,022, respectivamente. O **SAGC** proposto alcançou 1,919,

não superando somente **AG50**, por uma diferença de 10,28%. Em relação à cobertura, **SAGC** obteve o maior **NPS**, cobrindo 83,33% dos programas, enquanto **Best10**, **AG10** e **GA50** cobriram 40,00%, 66,67% e 80,00% dos programas **cBench**, respectivamente.

Considerando os programas do conjunto **Polybench**, **SAGC** obteve o maior desempenho médio entre as estratégias avaliadas, com **MGS** de 2,038. **Best10** alcançou 1,841, **AG10** 1,675 e **AG50** 2,008. A diferença da média de desempenho para **AG50** foi de apenas 3,02%, porém a cobertura obtida por **SAGC** foi de 92,86% dos programas **Polybench**, enquanto **AG50** cobriu 57,14%. **Best10** e **AG10** alcançaram cobertura de 53,57% e 42,86%, respectivamente.

Os resultados apresentados indicam que estratégias de compilação iterativa podem até conseguir maiores *speedups*, porém com alto tempo de resposta.

As métricas consideradas indicam que **SAGC** é uma boa estratégia para encontrar boas sequências de transformações para um programa de entrada, superando outras estratégias. Isso pois é a técnica que (1) forneceu bons resultados em um baixo tempo de resposta, (2) encontrou soluções baseadas nas características do programa de entrada e (3) utiliza conhecimento prévio para retornar uma sequência.

3.7 Trabalhos Relacionados

Cavazos et al. (2007) propuseram uma estratégia de aprendizagem de máquina que encontra sequências de transformações específicas para o programa de entrada. Em uma fase de treino, essa estratégia cria aleatoriamente diversas sequências para um grupo de programas treino. Depois da criação das sequências, sua estratégia coleta **PC** de cada programa treino. Com base nessas duas peças de informação, um modelo de logística de regressão é criado, o qual preverá a sequência. A fase de implementação coleta **PC** do programa teste e invoca o modelo de predição, e finalmente retorna seu melhor código alvo.

Os autores demonstraram que uma estratégia de aprendizagem de máquina é capaz de superar o melhor nível de transformação do compilador. Além disso, também indicaram a possibilidade de uso de **PC** como estratégia para mensurar a similaridade entre dois programas. A estratégia proposta nesta dissertação é similar a essa, pois também utiliza esquemas de aprendizagem de máquina, porém Cavazos et al. (2007) não avaliou outras formas de caracterização para medir a similaridade entre dois programas. Além disso, o **SAGC** descrito nesta dissertação não possui o custo da execução do programa para a coleta de **PC**.

Lima et al. (2013) propuseram o uso da estratégia RBC para encontrar sequências de transformações para cada programa de entrada. Os autores argumentaram que é possível encontrar boas sequências, com base em compilações prévias, para programas não conhecidos pelo sistema. Essa estratégia cria diversas sequências em uma fase de treino. Depois, em uma fase de teste, infere uma boa sequência para o programa teste baseada na similaridade entre o programa de entrada e cada programa da base. Lima et al. (2013) propuseram diversos modelos de medida de similaridade, também baseados em vetores que foram compostos por PC. Esse trabalho demonstrou que é possível inferir uma sequência que atinge múltiplos objetivos, como, por exemplo, eficiência no consumo de energia. Essa estratégia possui as mesmas limitações que o trabalho de Cavazos et al. (2007).

Tartara e Reghizzi (2013) propuseram uma estratégia de longo prazo, a qual o objetivo é eliminar a fase de treino. Nessa estratégia, o compilador é capaz de aprender, durante cada compilação, como gerar o melhor código alvo. De fato, eles propuseram um algoritmo evolucionário que cria diversas heurísticas com base em características estáticas do programa teste (Namolaru et al., 2010). Basicamente, essa estratégia realiza duas tarefas. Primeiro, extrai características do programa teste. Segundo, um algoritmo evolucionário cria heurísticas inferindo quais transformações devem ser ativadas. Esse trabalho utilizou uma boa estratégia estática para representar programas (CN), porém não formalizou a eficiência da representação.

O trabalho de Queiroz Junior e da Silva (2015) avaliou diferentes configurações de uma estratégia RBC, que visava encontrar sequências de transformações para programas de entrada. O objetivo desse trabalho foi avaliar o desempenho de sua estratégia utilizando: (1) diferentes base de dados; (2) diferentes coeficientes para identificar programas similares; e (3) diferentes caracterizações de programas. Apesar desse trabalho considerar a avaliação de diversas configurações, possui três problemas: (1) não descreve um formalismo para encontrar uma representação eficiente; (2) avalia somente duas representações; e (3) os resultados obtidos pelo seu sistema não utilizam somente uma configuração.

A Tabela 3.10 apresenta os trabalhos listados nesta seção e algumas de suas características, além das diferenças com a proposta apresentada neste capítulo. Vale ressaltar que tais trabalhos aplicam aprendizagem de máquina ao PST, porém não descrevem formalismos para avaliar a extração de conhecimento de programas.

Tabela 3.10: Comparação dos trabalhos relacionados com a proposta apresentada.

Trabalho	Estratégia	Caracterização	Diferenças
Cavazos et al. (2007)	Regressão Logística	PC	Paradigma de regressão logística
Lima et al. (2013)	RBC	PC	Não explora diferentes formas de caracterização
Tartara e Reghizzi (2013)	Aprendizagem Contínua	CN	Criação de novas soluções ao longo da utilização
Queiroz Junior e da Silva (2015)	RBC	PC, DC	Exploração de bases, não de formalismos de representação

3.8 Considerações Finais

Encontrar a melhor forma de representar o conhecimento depende de um determinado objetivo e requer avaliações detalhadas do formalismo construído.

Um problema complexo, no campo da ciência da computação, é gerar bons códigos alvo, pois é um problema dependente de programa. Isso indica que a estratégia proposta considera o programa durante a tomada de decisão. Além disso, precisa contemplar quais transformações devem ser aplicadas durante o processo de geração de código.

Apesar da literatura descrever diversas estratégias que tentam mitigar o PST, não há consenso em qual representação deve ser utilizada nestes sistemas. Ademais, diversas estratégias não consideram tal problema como dependente do programa, por causa da complexidade de identificar uma representação eficiente do conhecimento.

Este capítulo apresentou e validou uma representação eficiente do conhecimento para caracterizar programas, que pode ser extraída estaticamente e não é dependente de linguagens de programação ou arquiteturas de *hardware*. Outra contribuição é distinguir um coeficiente capaz de identificar programas que reagem similarmente quando compilados aplicando a mesma sequência de transformações.

Os resultados obtidos pelo **SAGC** demonstram que este é capaz de encontrar boas sequências de transformações, considerando o relacionamento entre as entidades do programa e suas características, além de superar outros sistemas de geração de código.

Sugere-se como trabalho futuro a investigação de modelos de recuperação de experiências anteriores, para assim saber qual a melhor maneira de recuperar as sequências dos programas mais similares nesta estratégia.

Estratégias para Seleção de Experiências Anteriores

Raciocínio Baseado em Casos (RBC) é um paradigma de resolução de problemas que considera experiências anteriores para tomar decisões para um problema de entrada, também considerado como uma sub-área da aprendizagem de máquina (Aamodt e Plaza, 1994). A intenção é resolver o problema baseado em um conhecimento prévio de uma situação similar, em um processo de aprendizagem, armazenando experiências passadas em uma base de conhecimento (Jiménez et al., 2011). Além de adaptar, pode combinar soluções para gerar uma nova, de acordo com situações similares (de Mántaras e Plaza, 1997).

No caso do problema de geração de código, o RBC pode ser aplicado de diversas maneiras, reutilizando conhecimento prévio de programas similares, selecionando bons algoritmos de transformação de programas similares para aplicar ao programa de entrada.

Este capítulo propõe e avalia diferentes estratégias para a recuperação de experiências anteriores em um sistema RBC aplicado à seleção de transformações em um sistema de geração de código.

As principais contribuições são:

- A proposta e avaliação de diferentes modelos de recuperação de experiências anteriores;
- A análise dos resultados da proposta, mostrando alternativas para melhorá-la;
- A avaliação experimental do melhor modelo proposto, de forma a confirmar a proposta.

4.1 O Paradigma RBC

Raciocinar baseando-se em experiências anteriores é frequente no pensamento humano, aplicado à solução de diversos problemas. Em um sistema RBC, uma situação anterior é capturada para aprender para futuros casos, reutilizando a solução aplicada. Dessa forma, RBC é um processo cíclico e integrado de resolução de problemas (Aamodt e Plaza, 1994).

O RBC pode ser dividido em quatro etapas: (1) recupera um caso de uma base de conhecimento por meio de uma medida de similaridade; (2) reutiliza a informação do caso similar para resolver o novo caso; (3) revisa o resultado para o novo caso, avaliando o sucesso da solução; e (4) retém a experiência útil para futuros casos. O Algoritmo 7 apresenta as etapas do RBC.

Algoritmo 7: Raciocínio Baseado em Casos.

Input: Novo Caso (c)
 Base de Conhecimento (BC)
Output: Solução (s)
 $c' \leftarrow$ caso similar à c em BC
 $s' \leftarrow$ solução aplicada à c'
 $r \leftarrow$ revisão de s' aplicada ao caso c
 $s \leftarrow$ aplicação de r ao caso c
 Anexar a experiência s à BC
return s

A construção da base de conhecimento é essencial para alcançar bons resultados em uma estratégia RBC, pois são as experiências contidas nela, que darão base à aplicação de novas soluções.

A medida de similaridade é responsável por encontrar o caso base entre todos os da base de conhecimento. Tal medida deve representar o problema, sendo precisa e caracterizar se as soluções aplicadas a casos anteriores podem ser aplicadas ao novo caso.

Tendo em vista os casos mais similares da base de conhecimento, é preciso saber como revisar as soluções aplicadas. Dessa forma, é preciso explorar modelos de recuperação desses casos, pois o resultado da aplicação de um sistema RBC pode variar de acordo com a revisão de suas soluções.

4.2 Modelos para Recuperação de Sequências

Um sistema de geração de código que utiliza RBC se baseia em experiências passadas para tomar a decisão de qual sequência de transformações aplicar ao programa de entrada.

Essa decisão é tomada por meio de uma medida de similaridade e um modelo de escolha em uma base de conhecimento.

Esse modelo de escolha tem como entrada uma lista ordenada de programas treino, que vão do mais similar da base de conhecimento até o menos similar. E a base de conhecimento contém boas sequências para cada programa, ordenadas pelo desempenho obtido.

Esta dissertação propõe analisar os seguintes modelos de recuperação de sequências:

- **RBC-ELITE**: Recupera todas as sequências presentes na base de conhecimento com desempenho acima do melhor nível de transformação do compilador, somente para o programa com maior similaridade.
- **RBC-UNICO**: Recupera N sequências da base de conhecimento, considerando por ordem de desempenho obtido, somente para o programa com maior similaridade.
- **RBC-PROX**: Recupera N sequências da base de conhecimento, considerando somente sequências que obtiveram desempenho acima do melhor nível de transformação do compilador. Se o programa com maior similaridade não possui N sequências melhores do que tal nível, o segundo programa mais similar é escolhido e suas sequências completam o teste, e assim por diante.

A principal questão envolvida na avaliação desses modelos trata da obtenção de desempenho (1) recuperando somente boas sequências do programa mais similar (**RBC-ELITE**); (2) recuperando somente sequências do programa mais similar, ordenadas por desempenho (**RBC-UNICO**); ou ainda (3) resgatando boas sequências de programas com similaridade próxima (**RBC-PROX**).

4.3 Instanciação do Sistema

O sistema RBC possui uma base de conhecimento, alimentada em uma fase *offline*, enquanto sua utilização se dá na fase *online*. As próximas seções descrevem essas fases.

4.3.1 Fase Offline

Esta fase consiste em uma fase de treino para modelos de aprendizagem de máquina. No modelo RBC, a fase *offline* consiste na construção da base de conhecimento, que utilizará os dados coletados para basear as decisões futuras do sistema.

Programas treino São compostos por programas retirados da suíte de testes da LLVM (Lattner, 2017), e do *The Computer Language Benchmarks Game* (Gouy, 2017). A Tabela 4.1 apresenta os programas treino.

Tabela 4.1: Programas treino para construção da base dos modelos RBC.

Suíte de testes da LLVM			
ackermann (T00)	flops-3 (T14)	mandel (T28)	queens-mcgill (T42)
ary3 (T01)	flops-4 (T15)	mandel-2 (T29)	quicksort (T43)
bubblesort (T02)	flops-5 (T16)	matrix (T30)	random (T44)
chomp (T03)	flops-6 (T17)	methcall (T31)	realmm (T45)
dry (T04)	flops-7 (T18)	misr (T32)	recursive (T46)
dt (T05)	flops-8 (T19)	n-body (T33)	reedsolomon (T47)
fannkuch (T06)	fp-convert (T20)	nsieve-bits (T34)	richards_benchmark (T48)
fbench (T07)	hash (T21)	oourafft (T35)	salsa20 (T49)
ffbench (T08)	heapsort (T22)	oscar (T36)	sieve (T50)
fib2 (T09)	himenobmtxpa (T23)	partialsums (T37)	spectral-norm (T51)
fdry (T10)	huffbench (T24)	perlin (T38)	strcat (T52)
flops (T11)	intmm (T25)	perm (T39)	towers (T53)
flops-1 (T12)	lists (T26)	pi (T40)	treesort (T54)
flops-2 (T13)	lpbench (T27)	queens (T41)	whetstone (T55)
<i>The Computer Language Benchmarks Game</i>			
binary-trees (T56)	fasta-redux (T58)	pidigits (T60)	
fasta (T57)	mandelbrot (T59)	regex-dna (T61)	

Sequências coletadas O espaço de busca da base de conhecimento precisa ser conciso para simplificar a procura pela solução do problema atual. Porém, a representatividade dos casos da base deve ser ampla, contendo boas soluções para vários tipos de problema.

As sequências de transformações são coletadas da seguinte maneira: (1) uma técnica de compilação iterativa é utilizada para coletar uma boa sequência de transformações para cada programa treino; (2) são adicionadas as 10 boas sequências encontradas por Purini e Jain (2013); e (3) adicionam-se os níveis de transformação -01, -02 e -03 da infraestrutura LLVM.

Cada programa treino é avaliado com todas as sequências coletadas, e o resultado dessa avaliação é gravado na base de conhecimento.

Técnica de compilação iterativa Um Algoritmo Genético (AG) é utilizado para coletar uma boa sequência para cada programa. A população inicial é aleatória, e cada indivíduo consiste em uma sequência de transformações, que é evoluída a cada geração. Dois indivíduos são escolhidos a cada iteração por meio da estratégia de torneio para gerar novos indivíduos por meio de um operador de *crossover*, além de uma mutação que pode ocorrer. O *crossover* possui probabilidade de 60% de ocorrência, enquanto a mutação possui probabilidade de 40%.

Os indivíduos iniciam com tamanhos de 1 até $|Espaço\ de\ Transformação|$, e assim os operadores podem ser aplicados a indivíduos de diferentes tamanhos.

A mutação pode modificar um indivíduo (1) inserindo uma nova transformação arbitrariamente; (2) removendo uma transformação da sequência em um ponto aleatório; (3) trocando duas transformações presentes na sequência em pontos arbitrário; e (4) alterando uma transformação da sequência por outra qualquer. Uma dessas modificações é realizada, escolhida de forma aleatória.

O GA é executado com 100 gerações, com uma população de 50 indivíduos, sendo que o indivíduo de melhor desempenho sempre permanece para a geração seguinte.

O critério de parada é a estagnação da solução por três gerações ou a aptidão menor do que 0,01 do melhor indivíduo.

Tal estratégia é similar à utilizada nos trabalhos de Purini e Jain (2013) e Martins et al. (2016).

Representação de programas As Características Numéricas (CN) especificadas no trabalho de Namolaru et al. (2010) são coletadas para cada programa treino, a fim de representá-los em uma comparação com o programa de entrada. Tais características foram produzidas sistematicamente por meio de experimentos que as geravam mensurando suas influências na aplicação de transformações de código.

4.3.2 Fase Online

A fase *online* consiste na utilização do sistema. Sendo um compilador, o sistema tem como entrada um código em uma linguagem fonte e como saída um código em uma linguagem alvo, que possui a mesma semântica do código de entrada. A Figura 4.1 apresenta o esquema de funcionamento do compilador baseado em RBC.

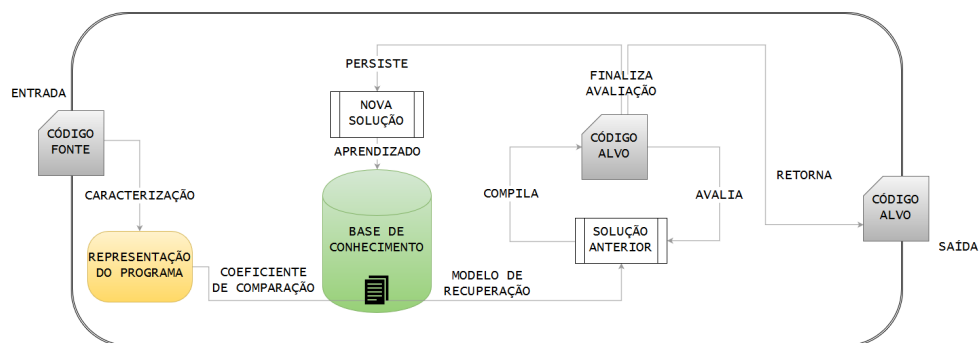


Figura 4.1: O sistema de geração de código baseado em RBC.

O fluxo de geração de código segue os seguintes passos:

1. **Caracterizar.** Primeiro o programa teste é caracterizado, assim o sistema extrai do programa um vetor de características com o qual irá representá-lo. A representação por meio das Características Numéricas (CN) propostas por Namolaru et al. (2010) foi escolhida para caracterizar o conhecimento sobre os programas.
2. **Comparar.** O conhecimento extraído, vetor de características, do programa teste é comparado com o acumulado em uma base de conhecimento, para assim obter soluções aplicadas a programas similares anteriormente compilados pelo sistema. Uma medida de similaridade compara a representação CN do programa de entrada com cada uma da base de conhecimento, por meio da medida Cosseno (CO), que é obtida por:

$$sim(P_x, P_y) = \frac{\sum_{w=1}^M (P_{xw} \times P_{yw})}{\sqrt{\sum_{w=1}^M (P_{xw})^2} \times \sqrt{\sum_{w=1}^M (P_{yw})^2}}$$

Na qual P_x e P_y são os programas a serem comparados e M é o número de características extraídas.

3. **Recuperar.** Após identificar programas similares que foram previamente compilados, o sistema recupera de tais programas boas sequências de transformações. Este processo estabelece um modelo de recuperação, no qual se define como serão aplicadas as soluções recuperadas, e em que ordem, considerando o grau de similaridade de cada programa da base de conhecimento. Os modelos de recuperação a serem avaliados são RBC-ELITE, RBC-UNICO e RBC-PROX.
4. **Compilar.** Obtendo a ordem de prioridade das soluções prévias, o programa teste é compilado com cada sequência de transformações recuperada gerando diversos códigos alvo.
5. **Avaliar.** Cada código alvo é avaliado de forma a encontrar a melhor solução, ou seja a melhor sequência de transformações pela qual se gerou o melhor código alvo. Dependendo do modelo de recuperação escolhido, pode haver um parâmetro N que especifica o número de avaliações a serem realizadas.
6. **Retornar.** O melhor código gerado é retornado ao usuário final.
7. **Aprender.** A base de conhecimento é alimentada com o novo conhecimento.

4.3.3 Aprendizagem Contínua

É importante que em um sistema baseado em conhecimento haja aprendizagem contínua ao longo do tempo. Para possibilitar tal tipo de aprendizagem, o RBC proposto utiliza um mecanismo de retroalimentação.

A retroalimentação pode funcionar de uma das seguintes maneiras:

1. persistindo somente o conhecimento das novas avaliações realizadas;
2. ajustando as soluções conhecidas da base de conhecimento para o programa teste;
ou
3. criando novas soluções ajustadas para o programa teste.

Persistir somente o conhecimento das novas avaliações realizadas do sistema faz com que não seja necessário realizar mais nenhuma avaliação após o retorno ao usuário. Essa solução é menos custosa ao sistema, pois não há compilações e avaliações adicionais.

Ajustar as soluções já conhecidas para o programa teste exige que se encontre a melhor solução possível da base de conhecimento para tal programa. Para isso, o custo de aplicar e avaliar todas as soluções conhecidas para o programa de entrada é adicionado ao sistema.

Criar novas soluções para o novo programa necessita que alguma estratégia de criação de novas sequências seja aplicada ao programa teste. Técnicas de compilação iterativa podem ser aplicadas ao código antes de este ser persistido na base. Dessa forma, poderiam ser criadas boas soluções, específicas para o novo programa, mas ao custo da execução dessa estratégia.

No RBC proposto foi optado por utilizar a segunda estratégia. Dessa forma, se tem por objetivo buscar armazenar o melhor conhecimento possível, tendo uma base de conhecimento específica, sem aumentar consideravelmente o custo do sistema em produzir novos conhecimentos.

4.3.4 Aprendizagem em Segundo Plano

A retroalimentação do sistema embora proporcione aprendizagem contínua, adiciona custos computacionais. Contudo, com a evolução das arquiteturas de computadores (Tanenbaum e Goodman, 1998), surgiram máquinas com mais de um núcleo de processamento (Patterson e Hennessy, 2013), o que permite a utilização de recursos de forma paralela. Assim, uma alternativa para reduzir tal custo do sistema é realizar aprendizado em segundo plano.

Após o sistema fornecer uma solução para o usuário, se tenta melhorar tal solução vasculhando outras possíveis solução da base do conhecimento. Tal processo, feito em segundo plano, atualiza a base de conhecimento após toda a base ser analisada.

O aprendizado em segundo plano utiliza recursos de hardware de forma transparente, não bloqueando a resposta ao usuário. Dessa forma, consegue-se ajustar as melhores soluções para o programa teste sem aumentar o tempo de resposta do sistema.

A Figura 4.2 apresenta o esquema de aprendizagem em segundo plano utilizada pelo RBC proposto.

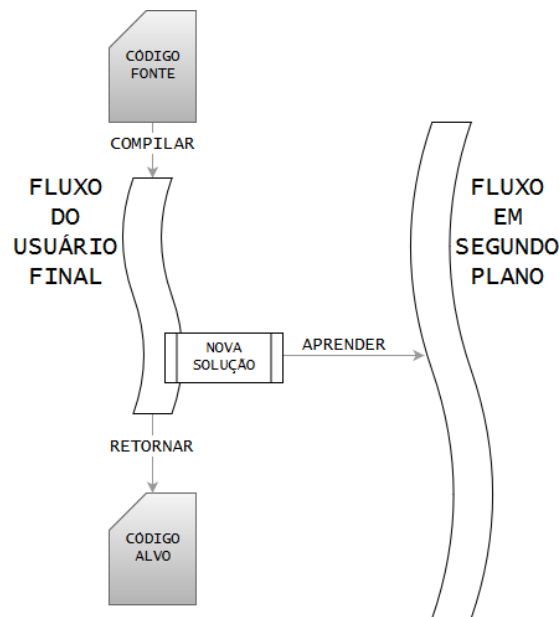


Figura 4.2: Aprendizagem em segundo plano, transparente ao usuário final.

Neste modelo, a responsabilidade pelo aprendizado sobre a nova solução passa para o fluxo que está em segundo plano, não fazendo uso dos recursos que estão visíveis ao usuário.

4.4 Configuração Experimental

Com o intuito de analisar os modelos de recuperação de RBC propostos, uma avaliação experimental foi realizada. A metodologia e os resultados são expostos e discutidos nas seções seguintes.

4.4.1 Ambiente

Arquitetura Intel(R) Core(TM) i7-3770 CPU 3.4GHz com 8GB RAM executando sistema operacional Ubuntu 14.04 x64 com *kernel* 4.2.0-41.

Compilador A infraestrutura de compilação LLVM 3.7.1¹ (Lattner e Adve, 2004).

Extração de características Implementou-se um módulo extrator de CN da representação intermediária da LLVM, durante o processo de compilação.

Tempo de Execução Cada programa foi executado 10 vezes para assegurar resultados precisos. Além disso, foram descartados 20% dos resultados: os 10% melhores e os 10% piores. Assim, a média geométrica do tempo de execução é calculada com base em 80% dos dados.

Programas A fase de testes utilizou programas que pertencem ao *Collective Benchmark* (cBench) (Fursin, 2017) e ao *Polyhedral Benchmark Suite* (Polybench) (Pouchet, 2017), apresentados na Tabela 4.2. Para os programas do cBench utilizou-se a entrada 1 e para o Polybench a entrada large.

Tabela 4.2: Programas teste utilizados na avaliação dos modelos RBC.

Programas cBench			
adpcm_c (C00)	dijkstra (C08)	pgp_d (C16)	susan_e (C24)
adpcm_d (C01)	ghostscript (C09)	pgp_e (C17)	susan_s (C25)
bitcount (C02)	gsm (C10)	qsort1 (C18)	tiff2bw (C26)
blowfish_d (C03)	jpeg_c (C11)	rijndael_d (C19)	tiff2rgba (C27)
blowfish_e (C04)	jpeg_d (C12)	rijndael_e (C20)	tiffdither (C28)
bzip2d (C05)	lame (C13)	rsynth (C21)	tiffmedian (C29)
bzip2e (C06)	mad (C14)	sha (C22)	
CRC32 (C07)	patricia (C15)	susan_c (C23)	
Programas Polybench			
2mm (P00)	covariance (P07)	gesummv (P14)	nussinov (P21)
3mm (P01)	deriche (P08)	gramschmidt (P15)	seidel-2d (P22)
adi (P02)	doitgen (P09)	heat-3d (P16)	symm (P23)
2mm (P03)	fdtd-2d (P10)	jacobi-2d (P17)	syr2k (P24)
bicg (P04)	floyd-warshall (P11)	lu (P18)	syrk (P25)
cholesky (P05)	gemm (P12)	ludcmp (P19)	trisolv (P26)
correlation (P06)	gemver (P13)	mvt (P20)	trmm (P27)

4.4.2 Metodologia

Os modelos RBC-PROX e RBC-UNICO foram parametrizados para recuperar 1, 3, 5 e 10 sequências inicialmente. O modelo RBC-ELITE não possui um parâmetro para o número de sequências recuperadas da base.

A avaliação utiliza cinco métricas para analisar os resultados:

¹<http://www.llvm.org>

1. MGS: Média geométrica de *speedup*;
2. NPS: Número de programas com *speedup* maior do que o nível mais agressivo de transformação (-03);
3. MLH: Melhoria em relação ao nível (-03);
4. NS: Número de sequências avaliadas; e
5. TR: O tempo de resposta da técnica.

O *speedup* é calculado como segue:

$$Speedup = \frac{Tempo_de_Execucao_Nivel_O0}{Tempo_de_Execucao}$$

Se optou para deixar retroalimentação e aprendizagem em segundo plano desabilitadas a fim de se encontrar os melhores modelos para implementação.

4.5 Resultados Experimentais

A Figura 4.3 apresenta os *speedups* alcançados para cada modelo de recuperação do RBC proposto.

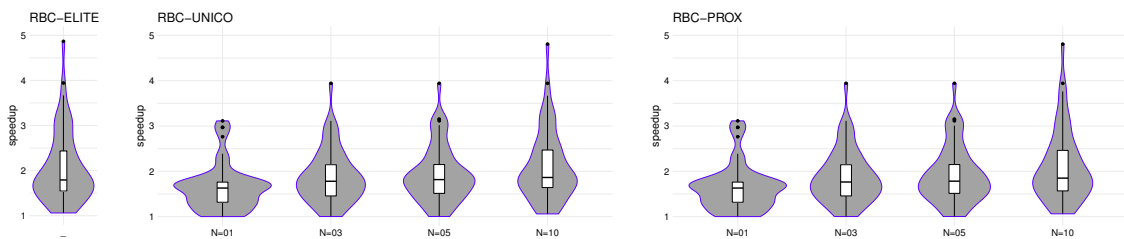


Figura 4.3: *Speedups* obtidos para cada modelo de recuperação.

MGS O maior *speedup* médio foi alcançado por RBC-UNICO.10, com o valor de 1,986, valor 1,05% maior do que os 1,976 alcançados por RBC-PROX.10 e 5,56% maior do que os 1,930 alcançados por RBC-ELITE. Com 3 e 5 casos recuperados, RBC-UNICO também foi superior a RBC-PROX, alcançando 1,842 contra 1,829 para N=05 e 1,796 contra 1,794 para N=03. Para N=01 os resultados foram próximos, com diferença de 0,3%, 1,456 e 1,453 para RBC-PROX e RBC-UNICO, respectivamente. Vale ressaltar que RBC-UNICO conseguiu superar o desempenho de -03 (1,838) com 5 casos recuperados.

NPS Para a métrica de cobertura, RBC-UNICO e RBC-PROX alcançaram valores parecidos. Enquanto RBC-UNICO alcançou 25,86%, 50,00%, 65,52% e 87,93%, RBC-PROX obteve 25,86%, 48,28%, 63,79% e 87,93% de programas cobertos para 1, 3, 5 e 10 experiências recuperadas, respectivamente. Por sua vez, RBC-ELITE cobriu 82,76% dos programas teste.

MLH A maior melhoria média em relação ao nível -03 foi alcançada por RBC-UNICO.10, com 18,55%, enquanto RBC-PROX.10 e RBC-ELITE melhoraram em média 13,55% e 17,72% os programas avaliados. Para N=1, ambos RBC-UNICO e RBC-PROX obtiveram melhoria média de 8,19%, enquanto para N=3 obtiveram 9,17% e 8,38%, respectivamente. Para N=5, RBC-PROX obteve melhoria média de 7,87%, enquanto RBC-UNICO melhorou os programas teste 9,79% em média. Para os programas não cobertos, RBC-UNICO.10 foi a configuração em que houve o menor declínio, apenas 1,08%, enquanto RBC-ELITE piorou em média 1,90% os programas e RBC-PROX.10 1,98%. Os outros valores de declínio para RBC-UNICO foram 58,87%, 22,27% e 22,45% para 1, 3 e 5 experiências recuperadas, respectivamente. Para RBC-PROX os valores foram 58,35%, 22,27% e 22,61% para os valores de N iguais a 1, 3 e 5, respectivamente.

NS RBC-ELITE é o único modelo dos 3 avaliados no qual não há parametrização do número de sequências avaliadas. Para esse modelo, o valor médio de NS foi 13,3, avaliando um número maior de sequências do que o valor mais alto para os modelos RBC-UNICO e RBC-PROX, com valor 10.

TR Apesar de avaliar um número maior de sequências, RBC-ELITE possui um tempo de resposta menor do que outros modelos, oferecendo uma resposta em 249,02s, enquanto para N=10, RBC-UNICO e RBC-PROX respondem em 311,57s e 320,82s, respectivamente. Para outros valores de N, RBC-UNICO respondeu em 41,36s, 100,77s e 159,72s para 1, 3 e 5, respectivamente, enquanto RBC-PROX possui TR de 41,28s, 105,28 e 168,89s para os mesmos valores de N.

Considerando os programas do conjunto cBench, o desempenho de RBC-PROX.10 foi o maior alcançado, chegando a 1,919, enquanto RBC-UNICO.10 obteve 1,892 e RBC-ELITE alcançou 1,866. Os outros valores para esse conjunto de programas foram: 1,649 para RBC-PROX.1, 1,788 para RBC-PROX.3, 1,831 para RBC-PROX.5, 1,642 para RBC-UNICO.1, 1,790 para RBC-UNICO.3 e 1,833 para RBC-UNICO.5. As maiores coberturas foram alcançadas por RBC-PROX.10 e RBC-UNICO.10, alcançando 83,33% dos programas cBench, enquanto RBC-ELITE obteve cobertura de 73,33%.

Para os programas *Polybench*, RBC-UNICO.10 obteve a maior média de *speedups*, com 2,092. Os valores para RBC-PROX.10 e RBC-ELITE foram 2,038 e 2,002, respectivamente. Os valores para 1, 3 e 5 experiências recuperadas foram 1,274, 1,799 e 1,826 para RBC-PROX e 1,274, 1,804 e 1,851 para RBC-UNICO, respectivamente. Quanto à cobertura, tanto RBC-UNICO.10 quanto RBC-PROX e RBC-ELITE cobriram 92,86% dos programas *Polybench*.

Os resultados apontam que, por uma leve diferença, RBC-UNICO alcança melhores resultados do que os outros dois modelos avaliados. Isso indica que é melhor recuperar sequências do programa mais similar com menos desempenho, do que recuperar de um programa não tão similar, mas que obteve sequências de maior desempenho. RBC-ELITE foi um meio de avaliar a recuperação de sequências tidas como boas apenas para o programa de maior similaridade, que avaliou sequências melhores (maior número de sequências avaliadas e menor tempo de resposta), porém não alcançou o desempenho obtido pelos outros modelos.

4.6 Melhores Soluções da Base

A base de conhecimento possui um número finito de soluções conhecidas que podem ser aplicadas ao problema de entrada. Em um sistema RBC, os resultados a serem alcançados dependem desse conhecimento acumulado. Tal base deve possuir boas soluções para os problemas de entrada, pois sem isso não importa quantos casos serão recuperados, não haverá bom desempenho. Desse modo, uma boa configuração RBC (representação do problema e coeficiente) deve recuperar o melhor caso possível que existe na base de conhecimento. Assim, esta seção tem por objetivo avaliar a distância entre o SGCA proposto e o melhor desempenho possível da base de conhecimento (*BestALL*²). Esses resultados podem indicar a consistência da configuração.

Os resultados escolhidos para comparação nesta seção são os melhores alcançados para cada modelo. A Tabela 4.3 apresenta os *speedups* alcançados por RBC-PROX e RBC-UNICO com $N=10$ e RBC-ELITE comparados a *BestALL*, na qual RBC-E = RBC-ELITE, RBC-P = RBC-PROX, RBC-U = RBC-UNICO.

Considerando somente os resultados obtidos para o conjunto de programas *cBench*, as diferenças médias para *BestALL* foram 9,91%, 7,04% e 4,59% para RBC-ELITE, RBC-UNICO e RBC-PROX, respectivamente. Para tais programas RBC-PROX alcançou não somente a melhor proximidade média, mas o maior número de programas com o mesmo desempenho de *BestALL*. Tal desempenho foi alcançado por RBC-PROX para 43,33% dos programas

²Compilar o programa utilizando todas as sequências de transformações presentes na base de dados

Tabela 4.3: Resultados do sistema de geração de código comparados a BestALL.

Programas cBench									
bench	RBC-E	RBC-U	RBC-P	BestALL	bench	RBC-E	RBC-U	RBC-P	BestALL
C00	1,678	1,747	2,476	2,740	C15	1,408	1,448	1,457	1,457
C01	1,371	1,372	1,567	1,764	C16	1,680	1,686	1,680	1,686
C02	3,668	3,668	3,668	3,668	C17	3,260	3,260	3,260	3,260
C03	1,812	1,812	1,823	1,834	C18	1,481	1,481	1,481	1,481
C04	1,752	1,752	1,780	1,786	C19	1,373	1,406	1,432	1,432
C05	1,505	1,505	1,505	1,505	C20	1,328	1,328	1,328	1,328
C06	1,957	1,981	1,960	1,981	C21	1,767	1,767	1,767	1,767
C07	1,069	1,106	1,112	1,112	C22	2,440	2,364	2,364	2,440
C08	1,262	1,262	1,262	1,272	C23	2,764	2,764	2,880	3,041
C09	1,061	1,061	1,063	1,063	C24	3,147	3,147	3,182	3,182
C10	2,302	2,302	2,302	2,302	C25	3,009	3,311	3,030	3,311
C11	2,009	2,009	2,009	2,009	C26	1,824	1,867	1,855	1,922
C12	2,053	2,053	2,053	2,056	C27	1,714	1,789	1,751	1,827
C13	2,331	2,261	2,261	2,331	C28	1,686	1,704	1,705	1,753
C14	2,084	2,084	2,084	2,099	C29	2,080	2,438	2,372	2,438
Programas Polybench									
bench	RBC-E	RBC-U	RBC-P	BestALL	bench	RBC-E	RBC-U	RBC-P	BestALL
P00	2,674	2,665	2,665	2,798	P14	3,000	2,600	2,600	3,000
P01	2,445	2,481	2,476	2,630	P15	1,753	1,753	1,753	1,868
P02	1,147	1,137	1,137	1,147	P16	2,128	2,126	2,126	2,128
P03	2,423	2,423	2,423	3,000	P17	1,315	1,315	1,315	1,315
P04	1,543	1,862	1,543	1,862	P18	1,573	1,808	1,809	1,831
P05	1,559	1,852	1,846	1,852	P19	1,620	1,874	1,873	1,893
P06	1,754	1,754	1,754	1,754	P20	1,872	2,704	2,028	2,704
P07	1,660	1,628	1,628	1,660	P21	3,221	3,221	3,764	4,160
P08	1,843	1,796	1,796	1,843	P22	1,160	1,160	1,161	1,161
P09	2,972	3,039	2,990	3,039	P23	1,774	1,780	1,780	1,793
P10	1,105	1,105	1,105	1,107	P24	1,574	1,574	1,574	1,574
P11	3,943	3,943	3,943	4,057	P25	2,583	2,583	2,583	2,583
P12	4,867	4,808	4,808	4,867	P26	1,429	2,000	1,429	2,000
P13	3,172	3,172	3,172	3,172	P27	2,172	2,303	2,172	2,303

cBench avaliados, enquanto para RBC-ELITE e RBC-UNICO foram 33,33% e 40,00% dos programas, respectivamente.

Para os programas Polybench, RBC-UNICO se aproximou 9,40% do desempenho médio de BestALL, enquanto RBC-ELITE obteve 17,21% de diferença e RBC-PROX 13,74%. Além disso, RBC-UNICO e RBC-ELITE conseguiram igualar o desempenho do melhor resultado da base em 39,29% dos casos, enquanto RBC-PROX alcançou tal valor para 21,43%, considerando o conjunto Polybench avaliado.

Em dados gerais, os resultados mostram que RBC-UNICO alcança uma média 6,75% menor de desempenho do que BestALL, enquanto RBC-PROX chega a 7,80% menos e RBC-ELITE 12,31%. Isso indica que a representação e o coeficiente que parametrizam a seleção das seqüências, principalmente com RBC-UNICO, provém resultados próximos dos melhores disponíveis.

RBC-ELITE obteve o mesmo desempenho de BestALL em 36,21% de todos os programas avaliados. Outros 31,03% dos programas obtiveram desempenho com uma diferença de

menos de 9,83% para o melhor possível da base, enquanto outros 27,59% variaram de 11,29% e 57,69%. Os últimos 5,17% obtiveram diferenças de 83,19% até 106,23%.

Para **RBC-PROX**, em 32,76% dos programas avaliados se alcançou o melhor desempenho possível. Além disso, para outros 41,38% o desempenho foi abaixo de 7,59% de distância para **BestALL**, e em outros 15,52% dos programas teste, a distância do desempenho variou entre 11,41% e 28,16%. Nos últimos 10,34% as distâncias variaram de 31,92% até 67,59%.

RBC-UNICO igualou o desempenho de **BestALL** em 39,66% de todos os programas avaliados, e para outros 36,21% a diferença para o melhor desempenho possível da base foi menos de 5,00%. Outras 13,79% das soluções obtiveram menos de 15,00% de diferença. As maiores diferenças se deram por conta de outros 6,90%, que variaram de 27,74% até 57,69%, e a discrepância dos últimos 3,45% (2 casos), que obtiveram distância de 93,89% e 99,28%.

4.6.1 Número de casos

O ideal é que se alcance o desempenho de **BestALL** com o menor número possível de casos anteriores recuperados, pois, quanto mais casos recuperados, mais avaliações de código (compilação + execução) são necessárias. Nesta seção se avalia a quantidade de avaliações necessárias para se obter o desempenho de **BestALL** em cada modelo de recuperação de experiências anteriores proposto.

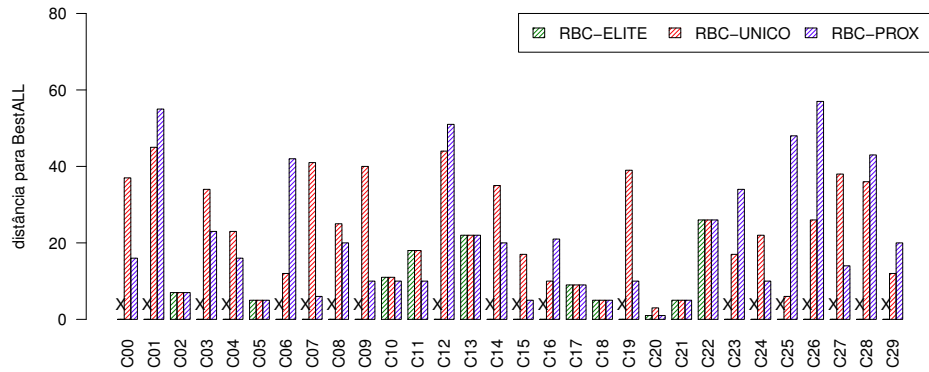
RBC-ELITE é um modelo que não parametriza o número de casos recuperados, resgatando o número de sequências que considera boas para o programa mais similar. Assim, há casos em que tal modelo não alcança o desempenho de **BestALL**, pois pode ocorrer da melhor sequência não ser considerada pelo modelo.

A Figura 4.4 apresenta os valores ideais de N em cada modelo e programa teste para se obter o valor de desempenho de **BestALL**. Nos casos em que **RBC-ELITE** não alcançou os valores de **BestALL**, o gráfico marca um “X” indicando onde estaria a barra com o valor.

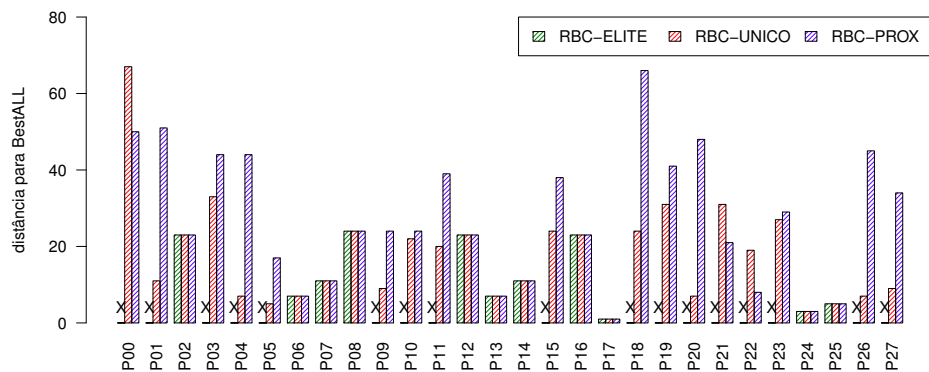
Para os programas **cBench**, as médias de casos recuperados para chegar ao melhor desempenho da base de conhecimento foi 10,90, 22,33 e 20,70 para **RBC-ELITE**, **RBC-UNICO** e **RBC-PROX**, respectivamente. Porém, **RBC-ELITE** atinge somente 33,33% dos programas avaliados.

Considerando os programas pertencentes ao conjunto **Polybench**, os valores da distância média de avaliações para alcançar **BestALL** foram 12,55, 17,54 e 27,18 para **RBC-ELITE**, **RBC-UNICO** e **RBC-PROX**, respectivamente. Contudo, a cobertura de **RBC-ELITE** foi 39,29%.

A média geral do número de casos avaliados para se chegar a **BestALL** confirma a superioridade dos resultados de **RBC-UNICO**, com 20,02 de média contra 23,83 de **RBC-PROX**.



(a) Programas cBench



(b) Programas Polybench

Figura 4.4: Número de experiências a serem recuperadas para alcançar o desempenho de BestALL

O modelo RBC-ELITE obteve média de 11,76 para os casos em que alcança BestALL, contudo tal alcance abrange somente 36,21% dos programas avaliados.

O desvio padrão das amostras foi 13,74 e 17,07 para RBC-UNICO e RBC-PROX, respectivamente. Isso significa que, embora os resultados do primeiro modelo sejam mais próximos à média, ainda há valores altamente discrepantes em sua amostra de resultados. Enquanto há o programa Polybench.P00 que necessita de 67 avaliações de sequências para chegar a BestALL, há Polybench.P17 que alcança tal objetivo no primeiro caso recuperado.

Com 10 casos recuperados (situação máxima avaliada na seção 4.2), **RBC-UNICO** alcançou o resultado de **BestALL** para 39,66% dos programas avaliados, enquanto **RBC-PROX** alcançou para 32,76%. Com 20 casos recuperados, **RBC-PROX** chega a 46,55% dos programas com o melhor desempenho possível, enquanto **RBC-UNICO** chega a metade dos programas (50,00%) com tal desempenho.

4.6.2 Média das Distâncias

Como exposto, o número de sequências avaliadas para se chegar ao valor de **BestALL** pode ser alto para alguns programas. Porém, há sequências que chegam a um desempenho bem próximo do melhor possível. A principal questão é o quanto é compensatório realizar mais avaliações em troca de maior desempenho. Quanto mais próximo de **BestALL**, mais próximo estará o melhor desempenho possível da base de conhecimento, porém quanto maior o número de avaliações, maior será o tempo de resposta do compilador.

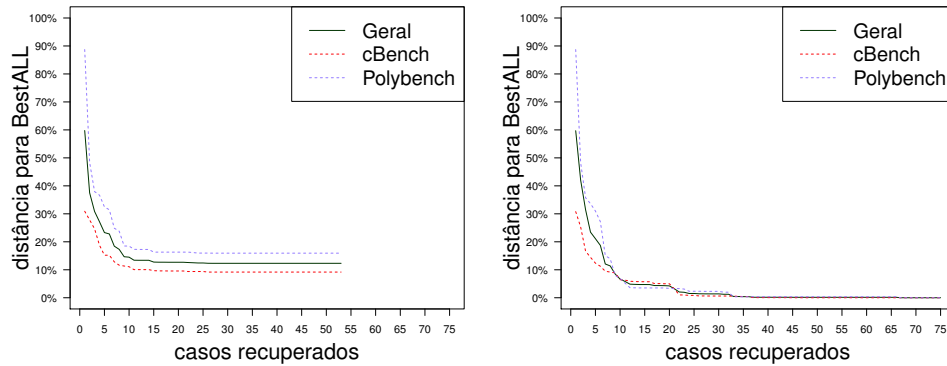
A Figura 4.5 apresenta os *speedups* médios obtidos após cada N casos recuperados. Para **RBC-PROX** e **RBC-UNICO** o valor de N vai de 1 até 75, que é o número máximo de sequências da base. Para **RBC-ELITE**, esse valor depende da quantidade de sequências com desempenho acima do melhor nível de transformação, que no caso dos experimentos realizados foi 53.

RBC-ELITE possui limitação quanto às sequências recuperadas, indo até 53 no caso máximo. Porém, para a maioria dos programas, o número de casos recuperados é menor, pois em média foram 13,3 casos por programa. Assim, esse modelo mantém uma distância de 12,31% para **BestALL** até para 53 casos recuperados.

Aproximações Significativas

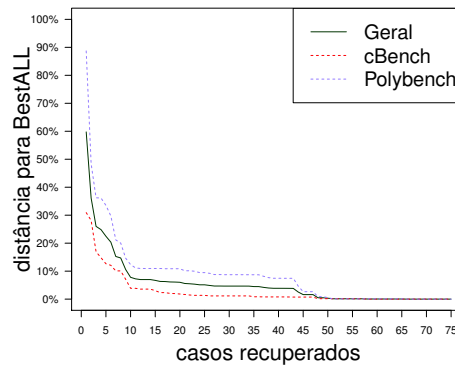
A distância para o desempenho de **BestALL** se inicia relativamente alta, obtendo, no geral, diferença de 59,77% para $N=1$. Porém, o desempenho se aproxima cada vez mais, a cada incremento do valor de N . Pode-se notar que, em alguns casos, a aproximação é mais significativa do que em outros casos. É importante destacar os pontos finais dessas aproximações, os quais são pequenas variações em relação ao número de avaliações que se aproximam do melhor desempenho consideravelmente.

Considerando Δ_x a distância para **BestALL** com x casos anteriores recuperados, os pontos finais de aproximações significativas considerados com N casos recuperados são os que obtiveram valor Δ_N mais de 15% menor do que Δ_{N-1} e que $N+1$ não seja um ponto de aproximação significativa, sendo ($5 \leq N \leq 75$) e ($\Delta_{N-1} \geq 1$). A importância de $N+1$ não ser considerado um ponto relevante está em se buscar o final da descida, no qual se



(a) RBC-ELITE

(b) RBC-UNICO



(c) RBC-PROX

Figura 4.5: Média das distâncias para BestALL conforme o número de casos anteriores recuperados.

obtem o menor valor de Δ . Além disso, são considerados valores de Δ_{N-1} maiores do que 1 para que em 15% de diferença haja um aumento significativo de *speedup*.

Para os programas *cBench*, os pontos relevantes de RBC-ELITE foram com 5 e 7 casos recuperados. De 4 para 5 casos recuperados, o valor de Δ cai 19,01%, obtendo *speedup* médio de 1,806, enquanto de 6 para 7 recuperações de casos anteriores a diferença para o melhor desempenho cai 15,55%, obtendo *speedup* de 1,829.

Ainda para o conjunto **cBench**, **RBC-UNICO** obteve pontos de aproximações significativas para os valores de N iguais a 7, 10 e 22, que obtiveram, respectivamente, declínios do valor Δ iguais a 15,78%, 27,33% e 67,68%.

O modelo **RBC-PROX** obteve tais pontos para 7, 10, 16, 26 e 35 casos anteriores recuperados, considerando os programas **cBench**. Para 6 para 7 recuperações, a diferença é de 16,51%, enquanto para Δ_{10} se obtém a maior diferença em relação à Δ_{N-1} : 45,51%. Os valores de tal diferença para N igual a 16, 26 e 35 foram 23,31%, 15,16% e 27,15%, respectivamente.

Considerando o conjunto de programas avaliados pertencentes ao **Polybench**, os pontos considerados foram para N igual a 7 e 9 com o modelo **RBC-ELITE**. Enquanto o declínio do valor Δ_{N-1} para 7 foi de 20,89%, para $N=9$ foi de 22,20%.

RBC-UNICO obteve os pontos 7, 9, 12, 24 e 33 como aproximações significativas para os programas **Polybench**. Esses pontos obtiveram descidas de 44,56%, 42,54%, 31,55%, 25,05% e 82,49%, respectivamente.

Ainda para os programas **Polybench**, o modelo **RBC-PROX** obteve aproximações relevantes nos pontos 7, 10, 45 e 48, os quais obtiveram, respectivamente, diferenças de 28,78%, 15,68%, 42,86% e 66,49% em relação à Δ_{N-1} .

Analisando a média geral entre os programas avaliados, os pontos de aproximações significativas para **RBC-ELITE** que prevalecem são 7 (19,12%) e 9 (15,19%), enquanto para **RBC-PROX** são os pontos 7 (25,11%), 10 (26,39%), 45 (36,27%) e 48 (69,48%). **RBC-UNICO** possui o maior número de pontos com desvios significativos considerando todas as avaliações, sendo 7 (35,49%), 10 (21,43%), 12 (16,75%), 22 (36,05%), 24 (21,71%) e 33 (62,41%).

Desempenho em Relação à BestALL

Pode-se notar que em **RBC-UNICO**, com $N \geq 12$ se obtém menos de 5% de diferença para **BestALL**, que diminui para menos de 2% com $N \geq 23$. Tal diferença vai para próximo de 1% com N maior do que 30. Porém, o desempenho de **BestALL** para todos os programas é alcançado somente com 67 sequências, por causa do programa **Polybench.P00**, que possui a maior número de casos recuperados para se alcançar o melhor desempenho.

Enquanto **RBC-UNICO** tem uma aproximação relativamente precoce com a média de **BestALL**, **RBC-PROX** mantém mais de 5% de diferença para tal desempenho com até 25 casos recuperados. O modelo de recuperação **RBC-ELITE**, por sua vez, não consegue se aproximar mais do que 12,31% de **BestALL**, independente do número de casos recuperados.

Tais análises continuam indicando a melhor forma de recuperar casos similares: sequências aplicadas ao programa mais similar, mesmo com desempenho inferior ao melhor nível de transformação.

Conforme o exposto, **RBC-UNICO** pode ser considerado o melhor modelo de recuperação de casos anteriores para o sistema de geração de código proposto por três motivos: (1) obteve o melhor *speedup* médio geral com menos de 10 avaliações; (2) obteve o maior número geral de aproximações significativas para as diferenças de desempenho, o que significa que poucas avaliações a mais fazem diferença no desempenho final; e (3) teve aproximação precoce de desempenho médio com **BestALL**, no que diz respeito a sequências avaliadas.

Em relação ao número de avaliações a serem realizadas, deve-se considerar tanto as aproximações significativas gerais do modelo quanto as marcas de diferenças para **BestALL** obtidas nessas curvas. Dessa forma, são escolhidos dois valores de N : 12 e 23. **RBC-UNICO** com 12 avaliações obteve menos de 5,00% de diferença para o desempenho de **BestALL**, e ainda foi o ponto final de uma aproximação relevante com 42,82% de diferença para a distância de **BestALL** para $N-3$ avaliações. O mesmo modelo com 23 casos recuperados é o primeiro ponto com menos de 2,00% de diferença para **BestALL**, além de ser o ponto seguinte de uma aproximação significativa e obter 53,91% de diferença para a distância de **BestALL** para $N-3$ avaliações.

4.7 Avaliação do Melhor Modelo

Com o intuito de avaliar o melhor modelo de recuperação isoladamente, um experimento para confirmar a eficiência da estratégia proposta foi realizado. Esta seção descreve este experimento, comparando a técnica proposta com uma estratégia da literatura e com os melhores resultados da base de conhecimento.

4.7.1 Ambiente Experimental

O ambiente experimental é o mesmo utilizado na Seção 4.4.1, exceto pelos programas teste utilizados. Os programas utilizados neste experimento fazem parte do conjunto de *benchmarks* SPEC CPU2006 (Henning, 2006), e possuem maior complexidade do que os pertencentes à cBench e Polybench. A entrada **train** foi utilizada, pois outras entradas do SPEC CPU2006 demandariam um tempo maior de experimentos. A Tabela 4.4 apresenta os programas utilizados.

Tabela 4.4: Programas do SPEC CPU2006 utilizados neste experimento.

Programas do SPEC CPU2006		
400.perlbench (S00)	445.gobmk (S06)	462.libquantum (S12)
401.bzip2 (S01)	447.dealII (S07)	464.h264ref (S13)
403.gcc (S02)	450.soplex (S08)	470.lbm (S14)
429.mcf (S03)	453.povray (S09)	471.omnetpp (S15)
433.milc (S04)	456.hmmmer (S10)	473.astar (S16)
444.namd (S05)	458.sjeng (S11)	483.xalanbmk (S17)

4.7.2 Metodologia

O modelo de recuperação de sequências avaliado é RBC-UNICO, escolhido com base nos experimentos anteriores. Considerando as distâncias para BestALL obtidas anteriormente, a avaliação se realizou com 12 e 23 casos anteriores recuperados para cada programa de entrada.

As métricas para avaliação são as mesmas utilizadas na Seção 4.4.2.

Para fins de comparação, a estratégia Best10 foi escolhida por apresentar um número de avaliações próximo ao da estratégia proposta. A técnica Best10 consiste avaliar o desempenho do programa aplicando as 10 boas sequências de Purini e Jain (2013), e retornar a que alcançou maior *speedup*.

4.7.3 Resultados e Discussão

A Figura 4.6 apresenta os desempenhos obtidos pelas estratégias Best10 e RBC-UNICO para cada programa do SPEC CPU2006.

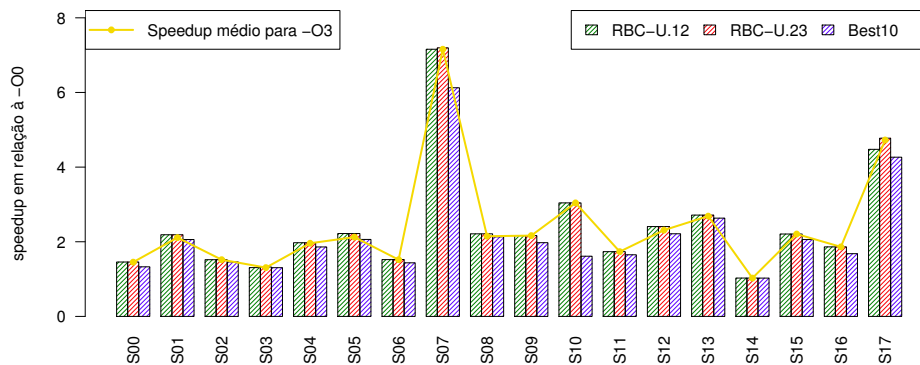


Figura 4.6: *Speedups* obtidos aplicando as estratégias comparadas

- MGS** O desempenho obtido pela estratégia RBC-UNICO foi superior a Best10, tanto com 12 casos recuperados, com MGS de 2,148, quanto com 23 sequências avaliadas, que alcançou *speedup* médio de 2,158. Best10 obteve um desempenho mais de 20,23% inferior do que RBC, com MGS de 1,944.
- NPS** Best10 não superou o desempenho de -03 nos programas avaliados. A estratégia RBC obteve 50,00% e 66,67% de cobertura para 12 e 23 casos recuperados, respectivamente.
- MLH** A estratégia proposta obteve melhoria média de 4,23% e 4,28% para os programas nos quais houve ganho sobre -03, para 12 e 23 casos recuperados, respectivamente. Houve somente um caso com perda de desempenho, para RBC-UNICO.12 com diferença de 24.98%, enquanto para RBC-UNICO.23 não houve perda. A estratégia Best10 não obteve melhoria em relação a -03 para os programas avaliados, enquanto seu declínio médio foi de 25,55%.
- NS** A quantidade de sequências avaliadas para a estratégia proposta foi parametrizada, sendo 12 e 23. A diferença de desempenho médio com 91% mais avaliações foi de 1,07%. A técnica Best10 avalia 10 sequências.
- TR** A execução dos programas deste experimento é mais demorada do que a execução dos programas da avaliação anterior. Isso se deve à complexidade dos *benchmarks* utilizados. Para 12 avaliações, o sistema demorou em média 2.369,87s, enquanto para 23 avaliações a média foi de 4.623,88s. Para Best10, a média foi de 2.570,62s. Nota-se que Best10 possui tempo de resposta maior do que RBC-UNICO.12, mesmo avaliando duas sequências a menos. Isso se deve ao fato de Best10 avaliar sempre as mesmas 10 sequências, enquanto a estratégia proposta avalia as 12 sequências que reconhece como mais adequadas ao programa de entrada.

4.7.4 Comparação com as Melhores Soluções da Base

Com o intuito de avaliar o desempenho obtido pela estratégia proposta, as próximas seções comparam os resultados alcançados com os melhores resultados presentes na base de conhecimento.

Desempenho obtido

A Figura 4.7 apresenta as diferenças dos desempenhos obtidos por RBC-UNICO.12 e RBC-UNICO.23 para o desempenho de BestALL.

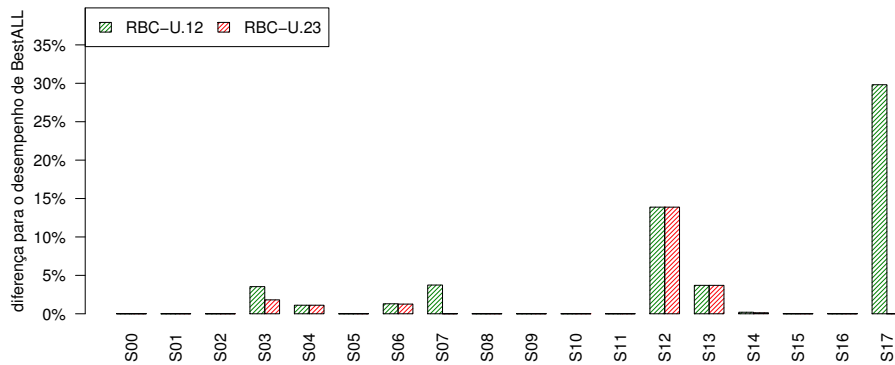


Figura 4.7: Diferença entre os desempenhos de RBC-UNICO e de BestALL.

Com 12 experiências recuperadas, a maior diferença para BestALL se deu para o programa S17, que ficou a uma distância de 32,77% do melhor desempenho da base. Com 23 casos anteriores recuperados, a maior diferença se dá para o programa S12, o qual ficou distante 13,88% do desempenho de BestALL.

A melhor solução possível foi alcançada para 50,00% dos programas avaliados, quando se recuperam 12 experiências anteriores. Em outros 38,89% a diferença de desempenho para BestALL variou de 0,04% até 3,74%. Os últimos 11,11% obtiveram tal diferença variando de 13,88% até 32,77%.

Para 23 experiências anteriores recuperadas, RBC-UNICO chegou ao desempenho de BestALL para 55,56% dos programas avaliados. Em uma fração de 38,89% a diferença para os melhores desempenhos da base variou entre 0,04% e 3,70%, enquanto que para 5,56% dos programas esse valor foi de 13,88%.

Pode-se notar que com 12 experiências recuperadas, o desempenho médio chegou a uma diferença de 2,26% com a melhor solução possível, enquanto com 23 recuperações tal diferença diminuiu para 1,26%.

Média das Distâncias

A Figura 4.8 apresenta a média de distância do desempenho obtido para cada valor de N.

Para os programas avaliados, com apenas 1 experiência anterior recuperada se obtém *speedup* médio de 1,894, a uma distância de 27,45% para o melhor resultado possível da base de conhecimento.

Os pontos que podem ser considerados com aproximação significativa para os resultados apresentados são com 7, 9 e 13 casos recuperados.

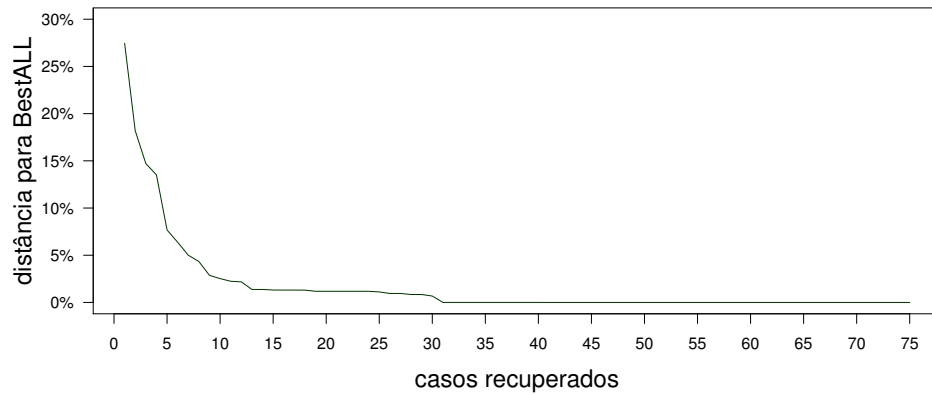


Figura 4.8: Distância média para o desempenho de BestALL.

Em comparação com a avaliação de 4 sequências, a diferença para o desempenho de BestALL diminui 62,96% com a recuperação de 7 sequências, indo de um *speedup* de 2,034 para 2,119.

Para 9 casos recuperados, se obtém *speedup* médio de 2,141 e o desempenho se aproxima 33,81% mais em relação ao melhor desempenho, quando comparado à avaliação de 8 casos, com *speedup* de 2,126.

Com *speedup* de 2,148 em 12 avaliações, se diminui a diferença relativa a BestALL em 36,90% com 13 avaliações, que alcança *speedup* médio de 2,156.

O desempenho da estratégia RBC-UNICO se aproxima para menos de 0,96% de diferença com BestALL com 26 avaliações de casos anteriores, e se iguala a BestALL com 31 avaliações.

A complexidade dos programas SPEC CPU2006 pode ser observada pela dificuldade em se obter desempenho pelas técnicas comparadas. Enquanto em experimentos anteriores RBC-UNICO alcançou 87,93% de cobertura para os programas cBench e Polybench com 10 avaliações de sequências, o máximo alcançado para o conjunto SPEC CPU2006 foi de 66,67%, com 23 avaliações.

Além da cobertura obtida, os *speedups* médios alcançados, mesmo para BestALL, foram próximos em relação ao nível de transformação -O3 para os programas SPEC, com melhoria de 3,57%. As mesmas sequências obtiveram, para os programas Polybench, 28,87% de melhoria em relação à -O3.

4.7.5 Uso de Aprendizagem Contínua e em Segundo Plano

Para avaliar o uso de aprendizagem contínua e em segundo plano é considerado que existe um tempo ΔT entre a compilação de dois programas, o qual representa o tempo de alimentar a base com conhecimento sobre o primeiro programa. Além disso, é importante perceber que a ordem de compilação dos programas pode influenciar o desempenho obtido, pois a base de conhecimento estará diferente a cada nova compilação.

Como não é possível determinar em qual ordem os programas serão compilados, este experimento considera que os programas são compilados em ordem alfabética.

Para facilitar a comparação com o sistema sem a retroalimentação, foram consideradas 12 avaliações para os programas SPEC CPU2006.

A Figura 4.9 apresenta os resultados do RBC, utilizando aprendizagem contínua e em segundo plano.

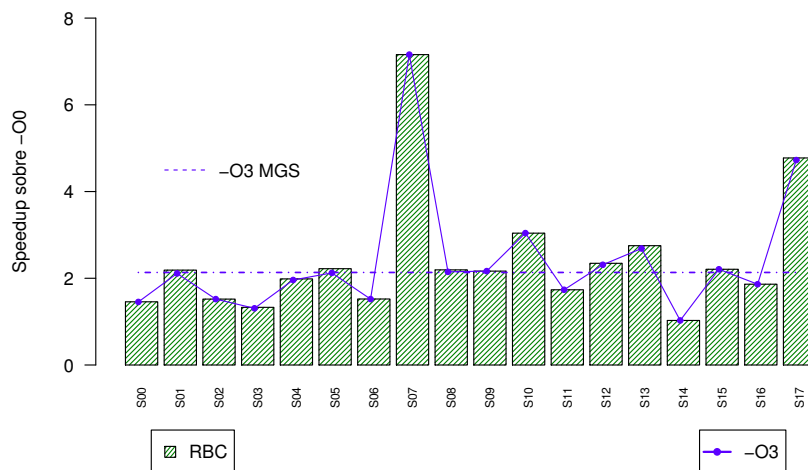


Figura 4.9: Speedup obtido pelo RBC com retroalimentação.

MGS A média de speedups obtida foi levemente superior (menos de 1%) comparada à de resultados sem retroalimentação, alcançando 2,155 de MGS.

NPS A cobertura foi 11,11% maior com retroalimentação, chegando a 61,11% dos programas avaliados.

MLH Apesar de conseguir melhoria para um número maior de programas, a melhoria média em relação à -O3 diminuiu para 3,80%, uma diferença de 0,3% para os resultados sem retroalimentação. Quanto à de perda de desempenho em relação

à -03, somente para o programa S10 se obteve 0,15% de perda, enquanto para o sistema sem retroalimentação também houve uma única perda, porém o percentual foi de 24,98% para o programa S17.

- NS O número de sequências avaliadas é parametrizado e foi utilizado como o mesmo do sistema sem retroalimentação com o menor N (12).
- TR O tempo de resposta do sistema chegou a diminuir em 3,23%, respondendo em média depois de 2293,37 segundos em cada programa. O tempo de resposta não aumentou devido à aprendizagem em segundo plano, responsável por ajustar as soluções da base de conhecimento para o novo programa de forma a não bloquear a resposta do usuário.

O sistema proposto mostrou-se efetivo para encontrar sequências de transformações que possibilitam o ganho de desempenho, mesmo para programas não vistos pelo compilador ao longo de seu treinamento e utilização.

O uso de compilação contínua mostrou-se atrativo para permitir o aprendizado contínuo do compilador, relacionando características dos novos programas compilados com as soluções presentes na base de conhecimento.

Além disso, a compilação em segundo plano permitiu que o tempo de resposta da estratégia continuasse o mesmo, não bloqueando a resposta do usuário para adquirir novos conhecimentos para o sistema.

Apesar da complexidade dos programas avaliados, a estratégia proposta se mostrou efetiva, conseguindo melhorias para os programas, com um número pequeno de avaliações. Porém, suas avaliações necessitam da execução dos códigos gerados, o que inclui um custo que não está no controle do projetista do compilador, e sim da entrada do usuário. Dessa forma, programas mais complexos demandam maior tempo de resposta, como foi o caso dos programas SPEC.

Para códigos complexos, a estratégia mostra-se adequada a aplicação em programas de dispositivos embarcados e de controle de rede, geralmente programas em que se busca maior desempenho. Deve-se encontrar um equilíbrio entre necessidade imediata do usuário e necessidade de desempenho do código gerado. Os tipos de programas citados tendem a realizar execuções com recursos limitados, tentando-se obter o máximo desempenho possível e assim existindo a possibilidade de se esperar um tempo maior para obter ganho de desempenho.

4.8 Trabalhos Relacionados

O trabalho de Mota et al. (2008) aplicou RBC à análise de imagens de satélite para identificar padrões de desmatamento de florestas. A medida de similaridade teve como base um classificador estrutural, que reconhece o padrão da imagem de entrada e a classifica estaticamente, recuperando a imagem mais similar de uma base de imagens previamente coletadas.

O trabalho de Jiménez et al. (2011) aplicou RBC para medir a qualidade de transmissão em redes ópticas, tendo uma base de conhecimento prévia gerada aleatoriamente e com distribuição uniforme em uma faixa de valores estabelecida. A recuperação do caso base leva em consideração parâmetros de um modelo de avaliação do fator de qualidade das redes ópticas, realizando a média ponderada do valor correspondente a cada parâmetro dos dados da rede de entrada.

Robertson e Watson (2012) aplicaram uma técnica RBC a um jogo de estratégia, baseando a escolha do caso base nas ações humanas de situações particulares que pudessem ocorrer. As análises das situações ocorriam em uma fase *offline*, para não diminuir a velocidade ou qualidade do jogo. A recuperação de casos base com retroalimentação do sistema permitiram a reação do jogo a uma gama de situações observadas.

O trabalho de Zhou e Wang (2014) utilizou o RBC para gerenciamento de desastres causados por tufões. Dados espaciais e geográficos compunham as informações coletadas e armazenadas na base de conhecimento, referentes a desastres anteriores. A soma ponderada de cada fator coletado é o cálculo para identificação do caso base, levando em consideração todos os parâmetros com relevante influência no resultado.

4.9 Considerações Finais

O sistema RBC proposto foi capaz de recuperar boas sequências de transformações para códigos de entrada. A avaliação das formas de recuperação de experiências indicou que, para este problema específico, é melhor recuperar experiências do caso mais similar do que de um caso não tão similar, mesmo que com melhor desempenho em suas soluções.

Além da recuperação de experiências, este trabalho comparou as melhores soluções da base de conhecimento com as obtidas em cada modelo. Tal comparação possibilitou nortear novas execuções da estratégia, buscando a melhor solução possível para o sistema.

A execução do modelo escolhido foi capaz de superar em 21,36% outra estratégia comparada, além de atingir uma cobertura de 66,67% dos programas avaliados, em relação ao nível mais agressivo de transformação da LLVM, alcançando melhoria média de até 4,28%

em relação a tal nível. Além disso, a estratégia se mostrou capaz de atingir resultados próximos aos melhores disponíveis na base de conhecimento com poucas avaliações.

Como trabalho futuro, sugere-se a investigação da relação das sequências da base de conhecimento com as características do programa de entrada, bem como o estudo das mudanças provocadas nessas características com a aplicação das transformações.

Conclusão

A necessidade por melhores desempenhos exige compiladores cada vez mais eficazes, que gerem códigos de maior qualidade. Isso faz com que projetistas implementem diversas transformações, mas a complexidade da relação entre transformações e código faz com que seja uma tarefa difícil determinar qual sequência obtém o melhor desempenho para um código de entrada. Avaliar todas as possibilidades é impraticável devido ao tamanho do espaço de busca. A aplicação de técnicas que demandam muito tempo para executar é inviável a usuários finais, o que faz com que diversos compiladores não implementem soluções para a seleção de transformações.

Esta dissertação apresentou estratégias para geração de códigos mais eficientes por meio da seleção de transformações. A primeira estratégia proposta foi a aplicação de uma metaheurística, compilando iterativamente o código de entrada. Tal estratégia alcançou resultados melhores do que outra técnica comparada, contudo possui um tempo de resposta alto para a aplicação a usuários finais.

Além da compilação iterativa, esta dissertação descreveu representações do conhecimento para caracterização de programas, apresentando uma técnica para avaliar tais representações. Essa avaliação indicou uma boa representação que é capaz de alcançar 81% de proximidade com os melhores resultados possíveis por meio de reações à aplicação de transformações. Uma aplicação da representação em um sistema de geração de código alcançou resultados 13,74% melhores do que o nível mais agressivo de transformação do compilador (-03) com um tempo 99% menor do que estratégias de compilação iterativa nas avaliações apresentadas.

A aplicação de uma representação de conhecimento em um sistema de geração de código se deu por meio do Raciocínio Baseado em Casos, um paradigma de aprendizagem

de máquina que toma decisões baseado em experiências anteriores. Este trabalho explorou diferentes formas de recuperação de experiências prévias em uma base de conhecimento, concluindo que, no geral, boas sequências para um programa de entrada são as aplicadas ao programa mais similar, independente do desempenho obtido por programas com menor grau de similaridade. A estratégia final apresentada foi capaz de obter cobertura de 66,67% em relação ao nível -03, superando em 20,23% o desempenho de outra estratégia escolhida para comparação.

A ativação da aprendizagem contínua foi capaz de melhorar os resultados para o conjunto de programas complexos no que diz respeito ao desempenho (pouco menos de 1% melhor) e à cobertura (mais de 11% superior).

O custo adicional da aprendizagem contínua não resultou em maior tempo de resposta ao usuário, devido ao aprendizado em segundo plano proposto para o sistema apresentado. O aprendizado em segundo plano é capaz de uma utilização mais eficiente dos recursos disponíveis na arquitetura utilizada, realizando suas tarefas de forma transparente ao usuário. Assim, nos experimentos com aprendizagem contínua e em segundo plano, o tempo de resposta chegou a ser menor do que quando tal estratégia foi desabilitada.

Comparada às técnicas de compilação iterativa, a estratégia proposta possui baixo tempo de resposta. Contudo, ainda são realizadas 12 ou 23 avaliações por meio da execução dos códigos gerados, com a aplicação de cada sequência de transformações recuperada. Tais avaliações exigem um custo que não é controlado, depende da entrada do usuário.

Para a técnica proposta ainda existe a necessidade de uma fase inicial de treinamento, que constrói a base de conhecimento. A estratégia depende da qualidade das soluções existentes nessa base, pois é dela que são recuperados os casos prévios aplicados ao código de entrada.

Comparado às técnicas anteriores da literatura, o tempo de resposta relativamente baixo aproxima a solução apresentada à utilização por usuários finais, obtendo desempenho próximo ao de estratégias que demandam horas de execução para encontrar uma boa sequência de transformações.

5.1 Trabalhos Futuros

Diversas questões podem ser aprimoradas a fim de maximizar o desempenho de programas, explorando a seleção de transformações. Pode-se ressaltar também pontos sobre a relação de transformações com estruturas de códigos, que auxiliariam na escolha das melhores sequências. Assim, propõem-se os seguintes trabalhos futuros:

Avaliar outras estratégias de seleção de transformações. Diversas estratégias podem ser aplicadas ao problema, podendo trazer resultados significativos e códigos gerados de boa qualidade.

Avaliar outras formas de caracterização de programas. Este é um caminho para aprimorar as estratégias de seleção de transformações por meio de aprendizagem de máquina. Além disso, diferentes medidas de similaridade podem ser avaliadas com o intuito de maximizar a captura da essência de cada programa.

Avaliação de outras estratégias de aprendizagem contínua. A aprendizagem contínua proposta consiste no ajuste das soluções presentes na base de conhecimento para um novo programa. Contudo, é possível utilizar outras formas de aprendizagem, como por exemplo aplicar técnicas de compilação iterativa. Embora técnicas de compilação iterativa possuam um alto tempo de resposta, o uso de compilação em segundo plano permite não adicionar esse custo ao tempo de resposta ao usuário final.

Construir base de conhecimento de outras formas. A base de conhecimento da proposta foi construída por meio de uma redução do espaço de busca que continha sequências conhecidas e resultados de estratégias de compilação iterativa. Além dessa, diversas formas podem ser exploradas, com diferentes tipos de programas treino.

Aprendizagem sem uma base inicial. Para evitar o custo da construção de uma base de conhecimento, sugere-se a aplicação de um sistema de geração de código que inicie sem uma base. Tal sistema deve permitir retroalimentação de dados, para que haja aprendizado contínuo ao longo da utilização do compilador. Assim, conforme receberia novos códigos, o sistema descobriria novas sequências de transformações e avaliaria o desempenho nos programas gerados.

Avaliar estaticamente os códigos gerados. A avaliação estática evitaria execuções de códigos gerados, o que diminuiria o custo do sistema. Além disso, o custo da avaliação seria fixo e não dependente da entrada do usuário. Desse modo, um caminho para realizar tal avaliação é gerar perfis das estruturas estáticas do código.

Recuperar sequência sem necessidade de avaliação. Para não haver avaliação do código gerado, apenas uma sequência deve ser aplicada pelo sistema de geração de código. A recuperação de apenas uma sequência necessita de uma maturidade no sistema de aprendizagem, que consiga gerar bons resultados sem ao menos uma

avaliação. Para tanto, é preciso levar ainda mais a fundo estudos sobre as estruturas dos programas e suas relações com as transformações de código disponibilizadas pelo compilador.

5.2 Considerações Finais

As estratégias propostas tiveram como objetivo mitigar a seleção de transformações para gerar códigos finais de maior desempenho, além de reduzir o tempo de resposta para aproximar a aplicação dessas estratégias a usuários finais de compiladores.

Pode-se considerar que os objetivos foram atingidos, já que bons resultados foram alcançados em um tempo inferior ao de outras estratégias da literatura.

Por fim, o problema abordado mostra-se amplo e complexo, sendo possível sua exploração por meio de diversas estratégias. Aprimorar a exploração da seleção de transformações possibilita gerar códigos de mais qualidade, o que acarreta uma utilização mais eficiente de recursos computacionais.

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.
- AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. 2 ed. Boston, MA, USA: Prentice Hall, 2006.
- 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.
- 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, 2011, p. 1–8.
- FOROUZAN, B.; MOSHARRAF, F. *Fundamentos da ciência da computação*. São Paulo: Cengage Learning, 2011.
- FURSIN, G. Collective Benchmark - Enabling realistic benchmarking and optimization. <http://ctuning.org/cbench>. Último acesso em 09/jan/2017, 2017.
- GOUY, I. The Computer Language Benchmarks Game. <http://benchmarksgame.alioth.debian.org/>. Último acesso em 09/jan/2017, 2017.
- 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.
- HENNING, J. L. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, v. 34, n. 4, p. 1–17, 2006.

- JIMÉNEZ, T.; DE MIGUEL, I.; AGUADO, J. C.; DURÁN, R. J.; MERAYO, N.; FERNÁNDEZ, N.; SÁNCHEZ, D.; FERNÁNDEZ, P.; ATALLAH, N.; ABRIL, E. J.; LORENZO, R. M. Case-Based Reasoning (CBR) to estimate the Q-factor in optical networks: An initial approach. In: *European Conference on Networks and Optical Communications*, 2011, p. 181–184.
- LATTNER, C. *LLVM: An Infrastructure for Multi-Stage Optimization*. Dissertação de Mestrado, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 2002.
- LATTNER, C. The LLVM Compiler Infrastructure. <http://llvm.org>. Último acesso em 09/jan/2017, 2017.
- 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, p. 75–86.
- 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.
- LIMA, E. D. *Soluções para o Problema da Seleção de Otimizações*. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, Universidade Estadual de Maringá, Maringá, PR, 2013.
- LIMA, E. D.; DE SOUZA XAVIER, T. C.; DA SILVA, A. F.; RUIZ, L. B. Compiling for performance and power efficiency. In: *23rd International Workshop on Power and Timing Modeling, Optimization and Simulation.*, 2013, p. 142–149.
- DE MÁNTARAS, R. L.; PLAZA, E. Case-based reasoning: an overview. *AI Communications*, v. 10, n. 1, p. 21–29, 1997.
- MARTINS, L. G. A.; NOBRE, R.; CARDOSO, J. A. M. P.; DELBEM, A. C. B.; MARQUES, E. Clustering-Based Selection for the Exploration of Compiler Optimization

- Sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, v. 13, n. 1, p. 8:1–8:28, 2016.
- MLADENVIĆ, N.; HANSEN, P. Variable neighborhood search. *Computers & Operations Research*, v. 24, n. 11, p. 1097 – 1100, 1997.
- MOTA, J. S.; CÂMARA, G.; FONSECA, L. M. G.; ESCADA, M. I. S.; BITTENCOURT, O. O. Applying Case-based Reasoning in the Evolution of Deforestation Patterns in the Brazilian Amazonia. In: *Proceedings of the ACM Symposium on Applied Computing*, Fortaleza, Ceara, Brazil: ACM, 2008, p. 1683–1687.
- MUCCI, P. J.; MOHAN, T. PAPIex - Command line/library utility to measure hardware performance counters with PAPI. <http://icl.cs.utk.edu/~mucci/papiex/papiex.html>. Último acesso em 09/jan/2017, 2017.
- MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- NAMOLARU, M.; COHEN, A.; FURSIN, G.; ZAKS, A.; FREUND, A. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In: *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, New York, NY, USA: ACM, 2010, p. 197–206.
- NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, v. 48, n. 3, p. 443 – 453, 1970.
- 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.
- PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011.

POUCHET, L.-N. Polybench: Polyhedral Benchmark suite. <http://www.cse.ohio-state.edu/~pouchet/software/polybench/>. Último acesso em 09/jan/2017, 2017.

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.

QUEIROZ JUNIOR, N. L.; DA SILVA, A. F. Finding Good Compiler Optimization Sets. In: *Proceedings of the 17th International Conference on Enterprise Information Systems*, Barcelona, Spain: SCITEPRESS, 2015, p. 504–515.

ROBERTSON, G.; WATSON, I. Case-based Learning by Observation: Preliminary Work. In: *Proceedings of The 8th Australasian Conference on Interactive Entertainment: Playing the System*, New York, NY, USA: ACM, 2012, p. 24:1–24:1.

SANCHES, A.; CARDOSO, J. M. P. On Identifying Patterns in Code Repositories to Assist the Generation of Hardware Templates. In: *International Conference on Field Programmable Logic and Applications*, Washington, DC, USA: IEEE Computer Society, 2010, p. 267–270.

SCHOLKOPF, B. *Learning With kernels - Support Vector Machines*. San Francisco, CA, USA: MIT Press, 2002.

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.

TANENBAUM, A. S.; GOODMAN, J. R. *Structured Computer Organization*. 4th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

TARTARA, M.; REGHIZZI, S. C. Continuous Learning of Compiler Heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)*, v. 9, n. 4, p. 46:1–46:25, 2013.

WU, Y.; LARUS, J. R. Static Branch Frequency and Program Profile Analysis. In: *Annual International Symposium on Microarchitecture*, New York, NY, USA: ACM, 1994, p. 1–11.

ZHOU, X.; WANG, F. A Spatial Awareness Case-Based Reasoning Approach for Typhoon Disaster Management. In: *IEEE International Conference on Software Engineering and Service Science*, 2014, p. 893–896.