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

TIAGO CARIOLANO DE SOUZA XAVIER

Solução Integrada para os Problemas de Seleção e Ordenação de Fase

Maringá
2014

TIAGO CARIOLANO DE SOUZA XAVIER

Solução Integrada para os Problemas de Seleção e Ordenação de Fase

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
2014

Dados Internacionais de Catalogação-na-Publicação (CIP)

Xavier, Tiago Cariolano de Souza
X3s Solução integrada para os problemas de seleção e ordenação de fase/ . -- Maringá, 2014.
105 f. il. : figs., tabs., color.

Orientador: Prof. Dr. Anderson Faustino da Silva.
Dissertação (mestrado) – Universidade Estadual de Maringá, Centro de Tecnologia, Departamento de Informática, Programa de Pós-Graduação em Ciência da Computação, 2014.

1. Compiladores - Otimização. 2. Seleção de fase – Aprendizagem de máquina. 3. Ordenação de fase – Ant system. 4. Algoritmo de colônia de formiga. I. Silva, Anderson Faustino da, orient. II. Universidade Estadual de Maringá. Centro de Tecnologia. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III Título.

CDD 22. ED.005.3
JLM001622

FOLHA DE APROVAÇÃO

TIAGO CARIOLANO DE SOUZA XAVIER

Solução integrada para os problemas de seleção e ordenação de fase

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

BANCA EXAMINADORA

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

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

Prof. Dr. Ademir Aparecido Constantino
Universidade Estadual de Maringá – DIN/UEM

Aprovada em: 27 de fevereiro de 2014.
Local da defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá.

AGRADECIMENTOS

Agradeço em primeiro lugar a meus pais, Joaquim e Izaura, pelo amor, carinho e apoio durante todos os anos de estudos. Mesmo nos momentos mais difíceis, eles foram as pessoas que sempre estiveram ao meu lado.

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

Faço um agradecimento especial aos professores Élvio João Leonardo e João Angelo Martini, que me orientaram no início do curso e que me deram a honra de trabalhar junto a eles. Principalmente o professor João, que não está mais entre nós, porém deixou um legado a todos que o conheceram.

Obrigado ao amigo Ewerton, pelos momentos em que trabalhamos juntos e também pelos momentos de descontração.

Agradeço a todos os colegas e professores do mestrado que contribuíram de diversas formas durante estes dois anos.

Por fim, gostaria de fazer um agradecimento especial ao meu orientador, o professor Anderson Faustino da Silva, que me proporciou valiosos ensinamentos que levarei para toda a vida. Muito obrigado pela oportunidade de trabalharmos juntos, pelo rigor, pela compreensão, pela amizade e por me ajudar a traçar novos caminhos.

Solução Integrada para os Problemas de Seleção e Ordenação de Fase

RESUMO

Compiladores modernos oferecem diversas otimizações para serem aplicadas ao código-fonte de um programa com o objetivo de aumentar o seu desempenho. Devido ao relacionamento complexo que as transformações possuem para cada diferente programa, descobrir qual o melhor conjunto de otimizações e qual a melhor ordem de aplicá-las sem intervenção humana são dois dos principais problemas enfrentados pelos projetistas de compiladores. Técnicas de compilação iterativa tentam mitigar estes dois problemas avaliando o desempenho do programa compilado com diversas sequências e escolhendo a melhor versão gerada. Bons resultados são obtidos, no entanto, na maioria dos casos a necessidade de um número extremamente grande de execuções da aplicação é um fator limitante. Abordagens que fazem uso de aprendizagem de máquina restrigem a quantidade de avaliações, porém usualmente não superam estratégias iterativas. Além disso, é escassa a pesquisa de estratégias que tentam integrar duas ou mais destas técnicas para mitigar o problema de selecionar e de ordenar otimizações ao mesmo tempo. Neste contexto, este trabalho apresenta uma estratégia para mitigação do problema da seleção de fase e outra para o problema de ordenação de fase, as quais fazem uso de uma estrutura de conhecimento prévio para selecionar e ordenar sequências. As duas abordagens foram implementadas em um *framework* de otimização de programas e, apesar de empregarem estratégias muito diferentes, são integradas para fornecer uma sequência de alto desempenho bem ordenada. Os resultados de experimentos conduzidos com os programas do SPEC2006 e cBench demonstraram que a abordagem integrada é capaz de otimizar estas aplicações consistemente.

Palavras-chave: problema de ordenação de fase. problema de seleção de fase. aprendizagem de máquina. clusterização. metaheurística. *ant system*. modelo de grafo. otimizações. compilador.

Integrated Solution For Problems of Phase Selection and Ordering

ABSTRACT

Modern compilers offer several optimizations to be applied to the source code of a program to order to increase its performance. Due to the complex relationship which transformations have for each different program, to find out the best optimizations set and the best order to apply them with no human interference are two hard problems which it is faced by compilers designers. Iterative compilation techniques try to mitigate these two problems evaluating compiled program performance with several sequences and choosing the best generated version. Good results are obtained, but in most cases requirement of a extremely big number of application executions is a limitative factor. Approaches use machine learning restrict quantity of evaluations, but usually they do not overcome iterative strategies. Besides, it is scarse research of strategies to try to integrate two or more from these techniques to mitigate problem of optimizations selecting and ordering at the same time. In this context, this work presents a strategy to mitigate phase selection problem and another one to phase ordering problem, which uses a prior knowledge structure to select and order sequences. Two approaches were implemented in a programs optimization framework and, despite they employ very different strategies, they are integrated to provide a well-ordered high performance sequence. Experiments results lead to programs from SPEC2006 and cBench demonstrated integrated approach can optimize that applications consistently.

Keywords: phase ordering problem. phase selection problem. machine learning. clustering. metaheuristic. ant system. graph model. optimizations. compiler.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

ACP: Análise de Componentes Principais
AG: Algoritmo Genético
BE: *Batch Elimination*
cBench: *Collective Benchmarks*
CE: *Combined Elimination*
CRC: *Ciclic Redundancy Check*
EM: *Expectation Maximization*
EEMBCv2: *Embedded Microprocessor Benchmark Consortium Version 2*
ES: *Exploratory Space*
FORTRAN: *Formula Translate System*
FSOF: *Framework* de Seleção e Ordenação de Sequências
GCC: *GNU Compiler Collection*
ICC: *Intel C++ Compiler*
IE: *Iterative Elimination*
JIT: *Just in Time*
LLVM: *Low Level Virtual Machine*
MOF: Módulo de Ordenação de Fase
MSF: Módulo de Seleção de Fase
NAS: *NASA Advanced Supercomputing*
PCVA: Problema do Caxeiro Viajante Assimétrico
PAPI: *Performance Application Programming Interface*
POF: Problema de Ordenação de Fase
PSF: Problema de Seleção de Fase
RI: Representação Intermediária
SPARC: *Scalable Processor Architecture*
SPEC2000: *Standard Performance Evaluation Corporation 2000*
SPEC2006: *Standard Performance Evaluation Corporation 2006*
SPECjvm98: *Standard Performance Evaluation Corporation Java Virtual Machine 1998*
SVM: *Support Vector Machine*
UTDSP: *University of Toronto Digital Signal Processing*

SUMÁRIO

1

Introdução

Compiladores modernos oferecem a possibilidade de aplicar otimizações (transformações) a um programa durante alguns dos estágios do processo de compilação. O objetivo deste recurso é criar uma nova versão semanticamente igual ao código de entrada, porém com desempenho superior. Alguns dos principais compiladores atuais, como GCC (*GNU Compiler Collection*)[1], LLVM (*Low Level Virtual Machine*)[2] e ICC (*Intel C++ Compiler*) [3], disponibilizam pelo menos algumas dezenas de otimizações para este fim, no entanto, fica à disposição do programador a decisão de escolher quais delas aplicar ao programa.

Esses compiladores também oferecem níveis (*flags*) de otimizações - normalmente chamados, O0, O1, O2 e O3 - que habilitam a aplicação de uma *sequência de otimizações* predefinida. Essas sequências são concebidas pelo projetista do compilador que, por meio de processos empíricos de avaliação de *benchmarks*, define esse conjunto ordenado de transformações, o qual pretende-se que melhore o desempenho para maioria dos programas.

Apesar desses níveis de otimizações melhorarem o desempenho de diversas aplicações, ao longo das últimas duas décadas pesquisadores têm descoberto que a qualidade do código transformado varia para cada programa (Cavazos e O'Boyle, 2006; Cooper et al., 1999; Fursin et al., 2005; Hoste e Eeckhout, 2008). Isso significa que embora a aplicação de uma sequência fixa possa melhorar o desempenho de um programa, ela não garante que o código resultante dele seja ótimo. Além disso, nem todos os programas compilados com

[1] http://gcc.gnu.org
[2] http://llvm.org
[3] http://software.intel.com/en-us/non-commercial-software-development

essa sequência terão um ganho de desempenho. De fato, o nome otimização pode soar enganoso, pois algumas otimizações podem modificar o código de tal forma que ocorra perda de desempenho, contrariamente ao pressuposto embutido no termo "otimização".

A principal dificuldade em descobrir qual a melhor sequência de otimizações para um programa específico reside no fato de que o relacionamento entre as diversas transformações é extremamente complexo e não é conhecido pelos projetistas e usuários de compiladores. Alguns trabalhos tratam do assunto fornecendo indicações de qual tipo de otimização aplicar antes ou após outra (Muchnick, 1997; Srikant e Shankar, 2007). Não obstante, ao se deparar com a imensa quantidade de possíveis combinações das sequências, sem uma estratégia adequada, é muito improvável para o usuário selecionar um conjunto de otimizações que forneça alto desempenho para o programa.

É conhecido que não apenas a escolha das otimizações, mas também a ordem na qual elas são aplicadas afeta o desempenho do programa compilado (Kisuki et al., 1999; Srikant e Shankar, 2007; Whitfield e Soffa, 1997). Nesse contexto, a pesquisa em otimização em compiladores trata dois problemas principais. O primeiro é o *Problema de Seleção de Fase*[4] (PSF), que corresponde a encontrar o conjunto de otimizações, sem considerar a ordem, que aplicado a um programa gera um código resultante ótimo. O segundo, chamado *Problema de Ordenação de Fase* (POF), se caracteriza por descobrir, dado um conjunto de otimizações, qual a melhor ordem de aplicá-las de modo que o programa resultante seja ótimo.

A maior dificuldade na resolução desses problemas está no tamanho do espaço de busca de sequências. Se um compilador fornece n otimizações para aplicar a um programa, o tamanho do PSF é 2^n, que corresponde a quantidade total de subconjuntos possíveis de serem gerados com n otimizações. Dado que o objetivo seja descobrir a melhor ordem de aplicação dessas n otimizações, o tamanho do espaço de busca de sequências do POF será $n!$, que é o total de permutações possíveis de serem formadas com n otimizações. Para um compilador como o GCC, que oferece 96 transformações possíveis de serem ativadas, o tamanho do espaço de busca do PSF e do POF seriam, respectivamente, $2^{96} = 7.9228162518 * 10^{28}$ e $96! = 9.916779349 * 10^{149}$. Note ainda que uma estratégia que tente solucionar os dois problemas, de maneira a fornecer o melhor conjunto de transformações dispostas na melhor ordem possível, se depararia com um espaço de busca maior do que 2^{96} ou $96!$.

Compilação iterativa é uma das estratégias mais usada ao longo das últimas duas décadas para mitigar esses problemas (Cooper et al., 2002). Nessa abordagem um mesmo programa é compilado com sequências diferentes e dentre todas as versões geradas a melhor

[4]Fase é um sinônimo para sequência de otimizações.

é escolhida. Devido à quantidade de sequências possíveis ser extremamente grande e à necessidade de compilar e avaliar o programa diversas vezes, algoritmos de compilação iterativa tentam percorrer o espaço de busca seletivamente. As técnicas para vasculhar esse espaço de busca de sequências podem ser classificadas em três categorias: busca parcial, algoritmos aleatórios e algoritmos genéticos.

Algoritmos de busca parcial tentam vasculhar uma porção do espaço de busca para encontrar sequências que forneçam melhor desempenho para o programa (Barreteau et al., 1999; Fursin et al., 2005; Gheorghita et al., 2005; Kisuki et al., 2000a, 1999; Kulkarni et al., 2006, 2009; Pan e Eigenmann, 2006). Algoritmos aleatórios ou estatísticos realizam essa busca empregando técnicas estatísticas e de aleatorização a fim de reduzir a quantidade de sequências avaliadas (Cooper et al., 2006; Haneda et al., 2005; Long e Fursin, 2005). Algoritmos genéticos se enquadram em um tipo de busca aleatória tendenciosa, em que conjuntos de transformações são combinados para gerar novas sequências, de modo que aquelas que apresentam melhor desempenho permanecem no processo, que continua iterativamente até que algum critério de parada seja satisfeito (Che e Wang, 2006; Cooper et al., 1999, 2002; Kulkarni et al., 2004, 2005; Kulkarni, 2007; Zhao et al., 2002; Zhou e Lin, 2012).

Apesar de apresentarem bons resultados, a principal fraqueza das estratégias de compilação iterativa é o tempo requerido para encontrar uma sequência de alto desempenho. Tipicamente um algoritmo necessita executar pelo menos algumas centenas de vezes o programa de entrada. Mais recentemente, o uso de técnicas de aprendizagem de máquina tem possibilitado a redução da quantidade de execuções do programa. Com essa abordagem um modelo preditivo é criado e utilizado para prever as melhores sequências de otimização para um programa. A criação do modelo requer muitas execuções, entretanto, esse processo ocorre durante a construção e implementação do compilador.

A literatura apresenta diversos trabalhos que fizeram uso de aprendizagem de máquina (Agakov et al., 2006; Cavazos et al., 2007; Cavazos e O'Boyle, 2006; Jantz e Kulkarni, 2013; Long e O'Boyle, 2004; Malik, 2010; Park et al., 2011; Purini e Jain, 2013; Thomson et al., 2010). No entanto, a grande maioria deles não trata o PSF e POF de maneira integrada.

Tendo como objetivo fornecer uma solução que possa mitigar conjuntamente os dois problemas, esta dissertação propõe uma estratégia de mitigação do PSF baseada em algortimos de clusterização e uma estratégia para mitigação do POF baseada em grafos.

A integração das duas abordagens é implementada em um *framework* de seleção e ordenação de sequências, o qual faz uso de uma estrutura de conhecimento prévio denominada espaço exploratório de sequências, a qual associa programas com sequências

de otimizações e serve como base de conhecimento para as duas abordagens. O *framework* aciona as duas estratégias para tentar descobrir um conjunto de otimizações, dispostas em uma dada ordem, que aplicadas no processo de compilação de um programa de entrada produz uma nova versão dele otimizada.

Além de fornecer essas duas abordagens, este trabalho também investiga a influência do espaço exploratório na descoberta de sequências. É descrito como a qualidade das sequências presentes em um mecanismo de conhecimento prévio como o que foi utilizado influencia nos resultados das estratégias que fazem uso dele.

Outros aspectos analisados correspondem à investigação: da influência dos algoritmos de clusterização no desempenho dos programas durante a estratégia de seleção; dos algoritmos propostos durante a ordenação das otimizações.

Os experimentos foram realizados com as suítes de *benchmarks* SPEC2006 e cBench. Para o SPEC2006 foi alcançado um ganho de *speedup* médio, sem considerar o programa com pico de desempenho, de até 05.77% e o melhor *benchmark* apresentou *speedup* de 2.89. Para o cBench o *speedup* médio da melhor configuração, sem considerar o programa com pico de desempenho, foi de 06.34% e o melhor *benchmark* alcançou um *speedup* de 56.04.

Resumidamente, as principais contribuições deste trabalho são:

1. Uma estratégia de apredizagem de máquina baseada em clusterização para mitigação do PSF.

2. Um algoritmo de transformação do problema de ordenação para um problema baseado em grafos, o qual considera o conhecimento prévio de informações de sequências e programas.

3. Uma abordagem de integração da estratégia de seleção e de ordenação para mitigação do PSF e POF de maneira integrada.

O restante do texto é organizado como descrito a seguir. O Capítulo 2 contextualiza os dois problemas alvos deste trabalho e apresenta o estado da arte dos trabalhos na litetura que propõem estratégias para mitigá-los. O Capítulo 3 apresenta o *framework* para otimização de programas, o mecanismo de conhecimento prévio e as duas abordagens propostas para o PSF e POF. O Capítulo 4 descreve a avaliação experimental para cada estratégia, os resultados obtidos, bem como uma investigação do espaço exploratório. Por fim, o Capítulo 5 apresenta as conclusões e discute alguns trabalhos futuros.

2

Referencial Teórico

Este capítulo descreve a estrutura de um compilador otimizante, define os principais problemas envolvidos na otimização de programas e apresenta os principais trabalhos da literatura propostos para mitigar esses problemas. Especificamente, a Seção 2.1 introduz o processo de otimização de programas realizada por compiladores; a Seção 2.2 apresenta a definição do PSF e POF; a Seção 2.3 aborda os principais tipos de soluções para os problemas; e a Seção 2.4 retrata dos principais trabalhos para mitigação do PSF e POF.

2.1 O Compilador Otimizante

Otimizações são algoritmos aplicados pelo compilador que transformam a representação do código-fonte sem alterar a sua semântica e que possuem como objetivo gerar um código-alvo mais eficiente. Existem dois tipos de otimizações: *análise* e *transformação*[1]. Uma otimização de análise serve para coletar informações do programa que esta sendo compilado, sem alterar a sua representação, e armazená-las para posterior consulta. Uma transformação modifica a representação do código para otimizar o programa e algumas vezes faz uso de informações coletadas pelas análises.

Um *compilador otimizante* é aquele que fornece um conjunto de análises e transformações durante o processo de compilação para otimizar o programa. A Figura 2.1 apresenta uma possível arquitetura deste tipo de compilador, a qual possui três etapas: *Front-End*, Otimizador e *Back-End*.

[1]Otimização e transformação são tratados na literatura como termos equivalentes. De fato, o termo transformação seria mais apropriado, pois uma otimização pode degradar o código em que é aplicada.

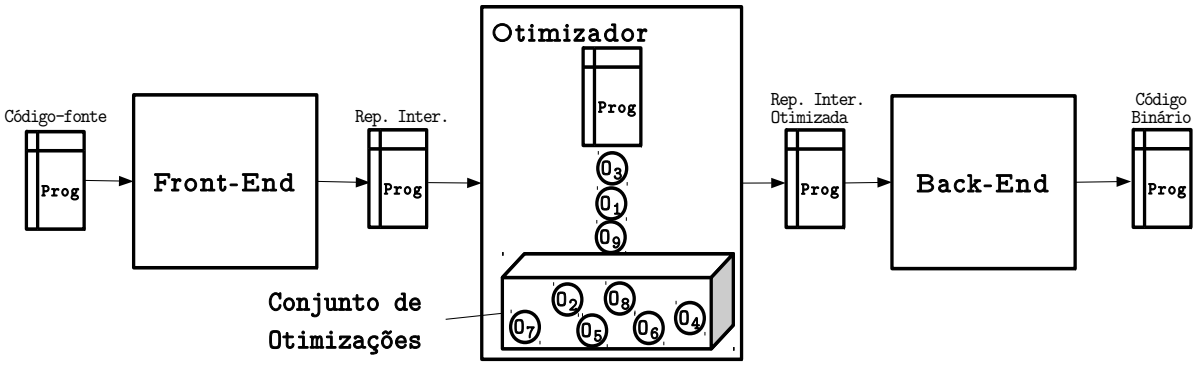


Figura 2.1: Arquitetura de um compilador otimizante.

O *Front-End* é a fase do compilador em que o código-fonte escrito pelo programador é submetido às etapas de análise e sua representação intermediária é gerada. No *Back-End* essa representação passa por um processo de transformação até que se gere o código binário (executável) para a arquitetura-alvo. Em um compilador tradicional apenas estas duas etapas estão presentes, no entanto em um compilador otimizante como o da Figura 2.1, entre o *Front-End* e o *Back-End* existe um componente adicional chamado Otimizador. Este componente, baseado na configuração fornecida pelo usuário ou em alguma estratégia do compilador, é responsável por aplicar uma sequência de otimizações à representação do código-fonte, gerando uma representação dele otimizada. Um subconjunto do conjunto de otimizações aplicadas pelo compilador é chamado de *sequência*[2] ou *fase*.

Idealmente, o principal objetivo do Otimizador é descobrir a função que mapeia sequências de otimizações para programas de maneira que o desempenho da aplicação seja ótimo. No entanto, um dos principais problemas para o Otimizador descobrir esta relação é que o efeito de se aplicar uma ou mais otimizações varia de programa para programa, o que pode ocasionar tanto o aumento quanto a diminuição do desempenho. Em outras palavras, cada programa possui um conjunto de otimizações específico ótimo (Cooper et al., 2002; Kisuki et al., 2000b, 1999).

Outro problema que o Otimizador precisa lidar é o fato de que os compiladores modernos oferecem dezenas de otimizações para otimizar o código, de modo que o espaço de busca se torna extremamente grande. Considere que para um compilador otimizante genérico C e um programa de entrada p o conjunto de otimizações disponíveis de C seja

[2]Ao se referir ao PSF, o termo sequência (ou fase) não indica a ordem de aplicação das otimizações. Ao se referir ao POF a ordem é considerada.

$$O = \{o_1, o_2, o_3, ..., o_n\} \tag{2.1}$$

O conjunto potência de O, $\wp(O)$, contém todos os subconjuntos possíveis de serem aplicados a um programa qualquer com as n otimizações. Como o tamanho de $\wp(O)$ é 2^n, esta é a quantidade de subconjuntos de otimizações que o Otimizador precisa avaliar para descobrir qual a melhor versão do programa compilado. Note que avaliar um subconjunto de otimizações consiste em compilar e executar o programa para inferir seu desempenho, o que é inviável mesmo para valores pequenos de n.

Além da questão da escolha de otimizações, como apontado por diversos autores (Kisuki et al., 1999; Kulkarni et al., 2009; Srikant e Shankar, 2007; Whitfield e Soffa, 1997) a ordem com que as elas são aplicadas também afeta o desempenho do programa, o que pode acarretar um número maior do que 2^n de sequências necessárias de serem avaliadas. Considerando esta questão, a próxima seção apresenta os dois principais problemas no contexto de otimização de compiladores que são os problemas alvos deste trabalho, os quais consideram a escolha e a ordem de otimizações.

2.2 Os Problemas de Seleção e Ordenação de Fase

A pesquisa em estratégias de compilador para otimizar programas tem sido dividida em dois problemas principais: Problema de Seleção de Fase (PFS) e Problema de Ordenação de Fase (POF). O PFS consiste em selecionar a melhor fase de otimização independente da ordem e o POF consiste em encontrar a melhor ordem para se aplicar as otimizações de uma fase.

Em notação matemática o PSF pode ser definido como a seguir. Considere uma sequência $s \in \wp(O)$ e a função de desempenho $\delta(p, s)$ que retorna o desempenho (tempo de execução, por exemplo) de se aplicar as otimizações de s em um programa p. O PSF pode ser definido como

- **Definição do PSF**: $\{s_1^* | \max_{\forall s \in \wp(O)} (\delta(p, s)) = \delta(p, s_1^*)\}$

onde max é a função máximo. O PFS consiste em encontrar dentre todas as possíveis sequências de $\wp(O)$, a sequência s_1^* que aplicada a p produz uma versão do programa com o desempenho máximo. Note que o programa compilado com as otimizações de s_1^* é ótimo com relação ao conjunto de otimizações O.

É importante notar que a Definição do PSF não considera a ordem, muito embora, na prática, o compilador aplicará as otimizações de s baseado em algum critério de ordenação.

De fato, alguns compiladores, por questões de dependência entre as otimizações, possuem uma ordem fixa, preestabelecida e que não pode ser alterada. No entanto, em compiladores em que a alteração da ordem é possível - como a LLVM utilizada neste trabalho - a não consideração da ordem pode ser conseguida por realizar uma geração aleatória das sequências de otimizações.

Para definir o POF matematicamente, primeiro é preciso recorrer à noção de permutação de otimizações, a qual é uma tupla de transformações em que a ordem é relevante. Dado o conjunto de otimizações O, a função $\pi(O)$ retorna todas as permutações dos elementos de O. Por exemplo, se $O = \{o_1, o_2\}$, então $\pi(O) = \{(o_1, o_2), (o_2, o_1)\}$, em que (o_1, o_2) é diferente de (o_2, o_1). O Problema de Ordenação de Fase para um programa p pode ser definido da seguinte forma

- **Definição do POF**: $\{s_2^* | \max_{\forall s \in \pi(O)} (\delta(p, s)) = \delta(p, s_2^*)\}$

O POF consiste em encontrar a permutação de otimizações s_2^* de O, que aplicada a um programa p possua o melhor desempenho. Note que todos os elementos de $\pi(O)$ possuem o mesmo tamanho $\mid O \mid$ e o que determina o desempenho de p é a ordem na qual as otimizações são aplicadas.

Por meio da definição dos dois problemas é possível verificar a complexidade envolvida em resolvê-los calculando a quantidade de vezes que a função max é computada em cada caso. Para o PSF a função máximo é computada para todos os elementos do conjunto $\wp(O)$, o qual tem tamanho 2^n. Já para o POF esta função é avaliada para cada elemento de $\pi(O)$, o qual é o conjunto de todas as permutações de O e tem tamanho de $\mid \pi(O) \mid = n!$.

Estes dois problemas são os mais proeminentes na área de otimização em compiladores, não obstante é interessante observar que a partir deles surgem algumas questões que geram novos problemas. Por exemplo, qual a melhor configuração de parâmetros das otimizações escolhidas? Uma sequência é capaz de aumentar o desempenho para todas as entradas de um programa?

A primeira questão está relacionada ao *Problema da Parametrização de Fase*, o qual consiste em selecionar a melhor configuração de parâmetros para todas as otimizações de uma fase de compilador. Diversos autores têm verificado que, dado uma sequência, um programa requer uma configuração específica das otimizações para gerar a sua versão ótima (Cooper e Waterman, 2003).

A questão que diz respeito às entradas de um programa está relacionada ao fato de que uma sequência pode fornecer desempenhos muito diferentes para o mesmo programa que ela foi aplicada, caso se modifique as entradas. O principal problema em usar várias entradas de dados para avaliar o desempenho de uma sequência, é que o tempo de avaliação aumenta conforme o tamanho do conjunto de dados. Em um estudo realizado por Fursin

et al. (Fursin et al., 2007), os autores executaram uma coleção de *benchmarks* com diversos conjuntos de dados, de maneira que algumas aplicações apresentaram variação de resultados mesmo quando compiladas com a mesma sequência. No entanto, os autores relatam que a variação não é grande o suficiente para invalidar o benefício de uma sequência.

Apesar dessas questões serem relevantes, devido à complexidade que trazem ao PSF e POF, elas são investigadas isoladamente. Por isso, quando tratando do problema de seleção ou ordenação de sequência, os parâmetros das otimizações são configurados para valores *default* e uma única entrada canônica é utilizada para cada programa.

A próxima seção descreve as principais técnicas para mitigar o PSF e o POF, bem como outros trabalhos relacionados.

2.3 Abordando os Problemas de Seleção e de Ordenação

Ao longo da última década vários autores tem proposto estratégias que almejam encontrar conjuntos de otimizações que superem as sequências fixas dos compiladores sem a necessidade de intervenção humana. Grande parte dessas abordagens são técnicas de compilação iterativa e aprendizagem de máquina, de modo que algumas são baseadas em outras estratégias como modelos de grafos.

Compilação iterativa na maioria das vezes apresenta melhores resultados do que aprendizagem de máquina, porém possui a desvantagem de frequentemente necessitar de um número excessivamente grande de execuções do programa.

Aprendizagem de máquina usualmente dispensa a necessidade de executar o programa do usuário muitas vezes e alguns algoritmos são capazes de apresentar desempenho similar à compilação iterativa. Nessa estratégia um modelo preditivo tenta predizer qual a melhor sequência para um programa. Apesar da criação do modelo ser custosa em termos de quantidade de avaliações de *benchmarks*, ela é feita durante a fabricação do compilador, não sendo visível para o usuário.

A seguir são apresentadas as abordagens de compilação iterativa e aprendizagem de máquina empregadas no PSF e POF.

2.3.1 Compilação Iterativa

Compilação iterativa consiste em avaliar alguns pontos do espaço de busca de sequências iterativamente. Essa avaliação é realizada pela aplicação de sequências a um programa e

execução dele para obter o seu desempenho. Um algoritmo de busca escolhe a próxima sequência considerando o desempenho das sequências aplicadas anteriormente.

A Figura 2.2 apresenta uma visão geral de um sistema de compilação iterativa. Um conjunto de transformações disponibilizadas pelo compilador é fornecido a um *Driver* que seleciona quais delas serão aplicadas ao programa de entrada. O *Driver* chama o *Compilador* que compila o código de entrada com a sequência escolhida. O código compilado é executado pelo *Executor*, que envia a informação de desempenho ao *Driver*, o qual mantém o registro de todas as sequências executadas durante o processo de compilação. Depois de um número predeterminado de iterações, o executável que obteve o melhor desempenho é fornecido como saída pelo *Driver*.

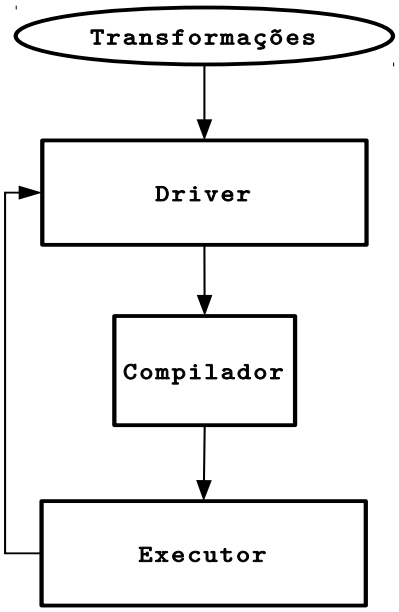


Figura 2.2: Compilação Iterativa. Adaptado de Kisuki et al.(Kisuki et al., 2000a)

A maneira como o *Driver* seleciona a próxima sequência é a parte crucial da abordagem e é condicionada por algum tipo de algoritmo de busca, o qual tenta procurar em uma parte do espaço de busca de sequências aquela que fornece melhor desempenho. É possível fazer uma classificação com três tipos de algoritmos de compilação iterativa: busca parcial, algoritmos aleatórios e algoritmos genéticos (AG).

Algoritmos de busca parcial possuem esse nome, pois utilizam heurísticas para vasculhar apenas uma parte do espaço de sequências. Geralmente essas abordagens se baseiam no conceito de vizinhança para migrar de uma sequência para outra entre duas iterações do algoritmo. A busca aleatória consiste em escolher sequências aleatoriamente ou por

meio de alguma técnica estatística. As duas abordagens podem fazer uso de mecanismos de poda do espaço de busca para reduzir a quantidade de sequências avaliadas.

Algoritmos genéticos são um tipo de busca aleatória tendenciosa largamente utilizada no problema de seleção ou de ordenação de fase por apresentar bons resultados. Essa estratégia tenta produzir boas sequências a partir de sequências que apresentaram bom desempenho anteriormente. O algoritmo começa com uma população de soluções (sequências) iniciais, chamada geração, que evoluirá. A cada passo todas as sequências geradas são avaliadas e as melhores soluções, determinadas por uma função de *fitness*, são combinadas por meio de um mecanismo denominado *crossover* para criar novas soluções, as quais formarão a próxima geração. Após algum critério de parada ou ao atingir um número limite de iterações, o algoritmo retorna a melhor versão do programa encontrada até aquele momento.

A principal desvantagem da compilação iterativa é o alto tempo para fornecer uma versão compilada do programa. Como o código-fonte precisa ser compilado e executado várias vezes, até que seja encontrada uma boa sequência, o tempo necessário para executar o algoritmo pode se tornar excessivamente grande. Apesar disso, bons resultados podem ser alcançados usando esse tipo de abordagem (Kulkarni, 2007; Pan e Eigenmann, 2006; Zhao et al., 2002).

2.3.2 Aprendizagem de Máquina

Aprendizagem de máquina foi aplicada ao PSF e POF com o propósito de ser uma abordagem viável com relação à quantidade de avaliações do programa. Em vez de compilá-lo iterativamente até encontrar uma sequência que aumente seu desempenho, o processo de compilação é dividido em uma fase de treinamento e outra de desenvolvimento. A Figura 2.3 ilustra um sistema de aprendizagem para o PSF ou POF.

Na fase de treinamento, um conjunto de *benchmarks* e conjuntos de otimizações são submetidos a um *Construtor*. Esse componente geralmente emprega alguma técnica de compilação iterativa para compilar e executar o conjunto inicial de programas com as sequências do espaço de busca e armazenar as informações resultantes. Por meio desses dados o *Construtor* gera um conjunto de exemplos que serão a entrada do *Algoritmo de Aprendizagem*. Este algoritmo é treinado com os exemplos e fornece como resultado um *Modelo*, o qual pode ser representado por diversos mecanismos (agrupamentos, árvores de decisão, redes bayesianas, etc), a depender do *Algoritmo de Aprendizagem*.

Na fase de desenvolvimento o novo programa a ser compilado fornece ao *Modelo* algum tipo de informação, geralmente dados que representam sua caracterização. O

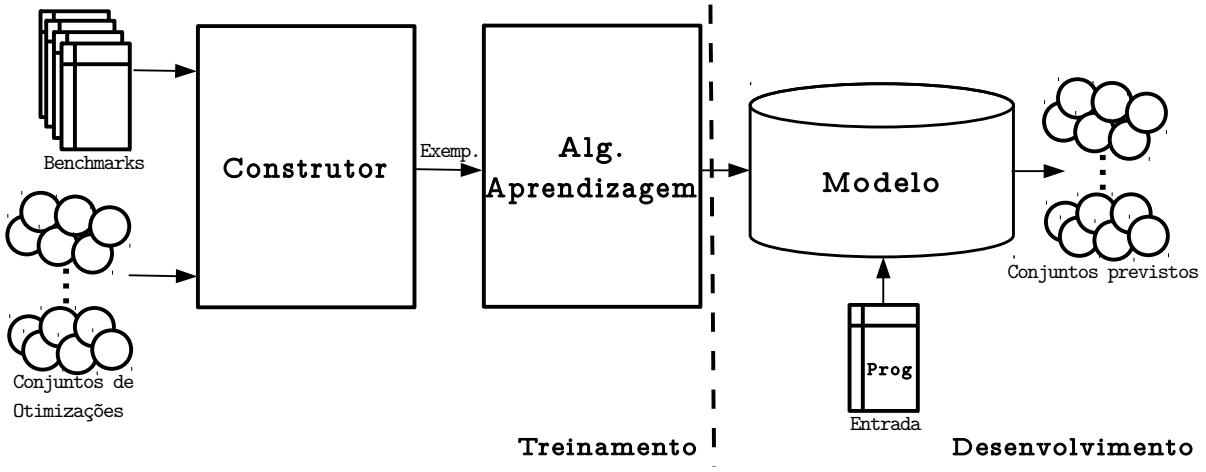


Figura 2.3: Aprendizagem de Máquina aplicada ao PSF ou POF. Adaptado de (Cavazos et al., 2007)

Modelo correlaciona o programa de entrada com as informações que o compõem e tenta prever ou gerar conjuntos que ao serem aplicados ao programa de entrada aumentarão seu desempenho. Todos os conjuntos gerados pelo *Modelo* são aplicados ao programa de entrada e a melhor versão do programa otimizado é fornecida ao usuário.

A fase de treinamento tipicamente é muito custosa com relação ao tempo requerido, pois cada *benchmark* inicial é executado centenas ou milhares de vezes. Isso faz com que a criação do modelo leve semanas ou meses para ser concluída. No entanto, note que esta fase ocorre *offline*, durante o momento em que os escritores do compilador estão realizando a implementação dele, de modo que não é visível ao usuário.

Além do próprio código-fonte, geralmente o principal tipo de informação fornecida pelos programas iniciais e pelo programa de entrada são suas caracterizações. Uma caracterização de programa corresponde a um conjunto de valores que representa o seu comportamento estático ou dinâmico. Caracterizações estáticas podem conter dados do código-fonte ou outra forma de representação estática do programa, como instruções *assembly*. Caracterizações dinâmicas correspondem aos valores dos contadores de desempenho, conhecidos como *performance counters*, os quais são disponíveis na maioria das máquinas modernas. Mais recentemente, novos tipos de caracterizações têm sido propostos baseados em estruturas do compilador, como por exemplo o grafo de controle de fluxo (Park et al., 2012).

2.4 Trabalhos Relacionados

Esta seção apresenta alguns dos principais trabalhos para mitigação do PSF e do POF, os quais estão divididos de acordo com a estratégia empregada. Uma análise comparativa é realizada para os trabalhos que possuem alguma semelhança com a estratégia proposta.

2.4.1 Compilação Iterativa

Busca Parcial

Pan e Eigenmann (Pan e Eigenmann, 2006) propuseram três algoritmos para procurar sequências de otimizações eficientes no espaço de busca: *Batch Elimination (BE), Iterative Elimination (IE)* e *Combined Elimination (CE)*. A ideia dos três algoritmos é descobrir e retirar otimizações nocivas de uma sequência *baseline*, a qual é constituída por todas otimizações do compilador. Esses algoritmos produziram excelentes resultados equivalentes a uma abordagem de busca exaustiva.

O *BE* compila um programa de entrada p com a sequência *baseline* (todas as n otimizações do compilador habilitadas), executa p e guarda seu desempenho D_{pb}. Em seguida, ele desabilita cada otimização i ($i = 1..n$) do *baseline*, uma a uma, compila o código com cada sequência gerada e guarda os desempenhos D_{pi}. A sequência final aplicada ao programa é formada pelo *baseline*, com todas otimizações i em que $D_{pi} < D_{pb}$ desabilitadas.

Uma limitação do *BE* é que ele não considera o efeito das interações entre as otimizações, pois elimina todas aquelas que apresentam pior desempenho do que a sequência *baseline* de uma só vez. O *IE* tenta abordar este problema desabilitando apenas a otimização que apresenta o pior desempenho. Inicialmente, o desempenho da sequência *baseline* D_{pb} e das sequências sem cada otimização i, D_{pi}, são obtidos como no *BE*. No entanto, apenas a otimização i com o menor desempenho D_{pi} é retirada. Esta sequência de tamanho $n - 1$ é agora o novo *baseline* e novamente o processo de descobrir a pior otimização é realizado com o novo *baseline* até que para toda otimização i do *baseline* $D_{pi} > D_{pb}$.

O *CE* é uma combinação das duas estratégias anteriores. Ele possui uma estrutura iterativa como o *IE*, porém a cada iteração ele tenta eliminar várias otimizações que produzem efeitos negativos ao mesmo tempo. Como o *BE* e *IE*, o *CE* avalia o desempenho do *baseline* sem cada uma das otimizações, no entanto, em seguida, o algoritmo cria uma lista ordenada de todas as otimizações nocivas. A otimização com pior desempenho, a qual é a primeira da lista, é retirada do *baseline*, que é atualizado. O novo *baseline* é

avaliado sem cada otimização restante da lista e todas aquelas que pioram o desempenho são retiradas. O processo se repete até que a retirada de quaisquer das otimizações não degrade o desempenho.

Pan e Eigenmann avaliaram sua estratégia com os programas do SPEC2000. Considerando os programas de inteiros e de ponto flutuante o percentual médio de melhoria foi de 8.15%. O *BE* tem complexidade $O(n)$ e o *IE* e *CE* têm complexidade $O(n^2)$. Esses valores delimitam a quantidade máxima de avaliações que cada algoritmo pode necessitar realizar.

Algoritmos Aleatórios e Genéticos

Cooper et al. (Cooper et al., 1999) projetaram um algoritmo genético para encontrar sequências do PSF que reduzem o tamanho do código de programas. 10 otimizações são disponíveis pelo compilador e o tamanho de cada sequência é de 12, o que totaliza um espaço de busca de tamanho 10^{12}. A função de *fitness* é o tamanho do código objeto produzido e o algoritmo produz 1000 gerações. Os resultados produzidos pelo algoritmo genético de Cooper et al. chegaram a reduzir o tamanho do código em 26% para alguns programas com relação a sequências fixas do compilador. Nesse trabalho, compilando todos os *benchmarks* com a mesma sequência, Cooper et al. puderam demonstrar experimentalmente que a sequência fornecida pelo algoritmo é melhor do que a melhor sequência fixa. Em outras palavras, eles verificaram que o desempenho de uma sequência difere de programa para programa.

Os *benchmarks* avaliados por Cooper et al. foram os programas da suíte FMM, escrita em FORTRAN, e alguns *kernels* de aplicação escritos em C. O algoritmo genético dos autores reduziu a quantidade e instruções estáticas em até cerca de 12.3% e instruções dinâmicas em até aproximadamente 2.27%.

O VISTA, proposto por Zhao et. al (Zhao et al., 2002), é um *framework* de ordenação de sequências de otimizações que faz uso de algoritmos genéticos e foi empregado no trabalho de Kulkarni (Kulkarni, 2007). O VISTA se diferencia de outras abordagens para o POF por permitir que o usuário selecione fatores de peso a serem considerados pela função de *fitness* do AG, tais como tamanho do código e tempo de execução. Kulkarni, além de analisar o POF por meio do VISTA, propôs algumas melhorias do AG para redução do tempo de busca. Essas melhorias consistem de algumas técnicas de poda do espaço de busca: o AG não executa sequências redundantes geradas pelos processos de mutação e *crossover*, pois guarda as sequências já experimentadas anteriormente; por meio de um checagem de CRC, um código já gerado anteriormente não é executado;

e por meio de detecção de código equivalente, programas não idênticos, porém que se conhece que possuem mesmo tamanho de código ou tempo de execução - por exemplo, dois programas que realizam as mesmas operações, porém com registradores diferentes - não são executados duas vezes.

Para doze *benchmarks* da suíte Mibench, em uma arquitetura SPARC, a implementação de Kulkarni foi capaz de melhorar o desempenho com relação à sequência padrão em cerca de 1% para o pior programa e aproximadamente 12.5% para o melhor. Além disso, as estratégias propostas foram capazes de reduzir o espaço de busca.

Recentemente um algoritmo genético multiobjetivo para o PSF, proposto por Zhou e Lin (Zhou e Lin, 2012), foi utilizado para melhorar o desempenho de tempo de execução e tamanho de código de sequências ao mesmo tempo. Nesse trabalho, a abordagem dos autores gera as soluções por meio de um AG e separa somente aquelas que melhoram os dois objetivos por meio de fronteira de Pareto. Elitismo é empregado no AG para melhorar a convergência do algoritmo para a fronteira de Pareto.

Essa abordagem de Zhou e Ling superou um algoritmo aleatório multiobjetivo e apresentou desempenho equivalente ao das sequências fixas do compilador com relação ao tempo de execução e *speedup*. O conjunto de *benchmarks* empregado foi o cBench, também utilizado neste trabalho.

A principal diferença entre as estratégias citadas nesta seção e a estratégia de seleção ou de ordenação propostas nesta dissertação, é que as últimas tentam explorar o conhecimento contido em um subconjunto do espaço de busca de sequências e de características de programas. O principal objetivo com essas abordagens é diminuir o *trade-off* existente entre a redução da quantidade de vezes que o programa necessita ser executado e o aumento de desempenho de tempo de execução por meio da exploração das informações existentes em um mecanismo de conhecimento prévio.

Outro ponto importante a ser observado é com relação aos *benchmarks*. Pan e Eigenmann otimizam os programas do SPEC2000, o qual é uma das coleções de programas mais relevantes na área de compilação. Isso se dá pois o SPEC2000 é constituído de aplicações reais, de uso científico e que são capazes de estressar os sistemas de memória e processamento. Por esse motivo o trabalho de Pan e Eigenmann (Pan e Eigenmann, 2006) é importante, pois consegue um alto desempenho para um conjunto de programas significativo. Os outros trabalhos avaliam *kernels* de aplicações ou suítes como o Mibench, o qual é importante por conter aplicações da área de sistemas embarcados, porém tem a desvantagem de possuir aplicações de tempo de execução muito curto, característica que muitas vezes prejudica avaliações experimentais consistentes. O cBench utilizado no trabalho de Zhou e Lin também é uma coleção de programas relevantes no contexto de

seleção de otimizações, pois apesar de possuir os mesmos programas do Mibench, suas aplicações foram modificadas para que tenham um tempo de execução maior.

Assim como os trabalhos de Pan e Eigenmann e Zhou e Lin, os *benchmarks* avaliados nesta dissertação são relevantes, a saber, o cBench e principalmente o SPEC2006, o qual é a versão posterior do SPEC2000.

2.4.2 Aprendizagem de Máquina

Clusterização

A abordagem de aprendizagem de máquina proposta neste trabalho é baseada em um algoritmo de clusterização. Este tipo de algoritmo tem sido adotado em duas pesquisas recentes de maneiras distintas para mitigar o PSF. A seguir, os trabalhos de Thomson et al. (Thomson et al., 2010) e Purini et al. (Purini e Jain, 2013) são brevemente apresentados e os aspectos mais relevantes e as principais diferenças com relação ao trabalho atual são apontados.

No trabalho de Thomson et al. (Thomson et al., 2010), os autores reduziram a quantidade de execuções e tempo requeridos na fase de treinamento de um algoritmo de aprendizagem por meio de clusterização de programas. A fase de treinamento consiste em aplicar sequências escolhidas aleatoriamente a um conjunto de *benchmarks*. Para reduzir a quantidade de programas necessários nessa etapa, um algoritmo de clusterização cria grupos de programas semelhantes, de modo que dentro de cada grupo um *benchmark* é escolhido para representar todos os demais. Apenas esses *benchmarks* representativos são empregados nesta fase inicial do ambiente de compilação.

Para realizar o agrupamento, inicialmente o algoritmo extrai de cada programa do conjunto inicial um vetor de características, em que cada característica consiste de uma instrução *assembly*. Cada instrução *assembly* corresponde a um eixo em um espaço multidimensional de programas, e cada vetor de características representa um programa (ponto) nesse espaço multidimensional. Devido ao grande número de instruções *assembly* disponíveis, uma técnica de redução de dimensionalidade é aplicada para reduzir a quantidade de elementos de cada vetor.

O algoritmo de clusterização *GustafsonKessel* é aplicado no espaço multidimensional formado pelos vetores de características, a fim de identificar os grupos de programas semelhantes. Uma técnica simples de vizinho mais próximo é empregada para escolher de cada grupo criado, o programa mais próximo ao centro do *cluster* (centroide). Esses programas-centroides passam então a representar seus respectivos grupos, pois correspondem ao ponto (programa) mais próximo de todos os outros programas do grupo.

Apenas os programas-centroides participam da fase de treinamento, de maneira a reduzir a quantidade de execuções e o tempo requerido nesta etapa.

No trabalho de Thomson et al., de um conjunto inicial de 44 programas, 6 foram utilizados na fase de treinamento. Para a avaliação experimental foi empregado o conjunto de *microkernels* EEMBCv2, de modo que o *speedup* médio foi de 1.14.

O algoritmo de clusterização usado por Thomson et al. teve como propósito reduzir a quantidade de programas na fase de treinamento. No trabalho desta dissertação isso não foi um objetivo, pois esta fase ocorre *offline*, durante a fabricação do compilador, não é visível ao usuário e pode ser realizada em um tempo satisfatório de poucas semanas. O objetivo do algoritmo de clusterização proposto neste trabalho é descobrir o grupo de programas, provenientes do mecanismo de conhecimento prévio, que mais se assemelham a aplicação de entrada e aplicar a melhor sequência de otimizações deles ao novo programa. Neste o algoritmo de clusterização é executado para cada novo programa, durante a fase de desenvolvimento, de maneira a encontrar o grupo no qual ele será inserido, enquanto que no trabalho de Thomson et al. o algoritmo é executado na fase de treinamento para escolher os programas-centroides. A Seção 4.3.3 demonstra que a estratégia de clusterização proposta neste trabalho alcança maiores *speedups* do que a dos outros autores.

Outra diferença das duas estratégias é com relação a caracterização dos programas. Em vez de caracterizá-los com informações estáticas do código-fonte, tal como instruções *assembly*, neste trabalho a caracterização de cada programa se dá por meio de informações dinâmicas de tempo de execução, *performance counters*. Esse tipo de caracterização foi utilizada em outros contextos (Cavazos et al., 2007; Park et al., 2011) e a investigação neste trabalho consiste em verificar como essas informações dinâmicas de tempo de execução podem influenciar a caracterização de programas especificamente para um algoritmo de clusterização.

É importante ressaltar que a coleção de *benchmarks* avaliada por Thomson et al. é composta por *microkernels* de aplicações de sistemas embarcados semelhantes ao Mibench. Por outro lado, a avaliação experimental deste trabalho é realizada com o SPEC2006, uma das mais relevantes suíte de *benchmarks* atuais. Além disso, os autores utilizam os mesmos *benchmarks* na fase de treinamento e desenvolvimento e empregam a técnica de *leave one out* para realizar a avaliação. Em vez disso, a fase de seleção deste trabalho possui dois conjuntos distintos, um para treino e outro para teste. Essa última abordagem evita a possibilidade de treinar e testar aplicações de uma mesma coleção de *benchmarks*, as quais usualmente são compostas por programas similares, o que pode influenciar nos resultados obtidos.

Purini e Jain (Purini e Jain, 2013) propuseram uma estratégia que consiste em descobrir um conjunto pequeno de boas sequências. Cada sequência desse conjunto reduzido é boa para uma classe de programas, de modo que um novo programa de entrada, o qual possivelmente pertence a alguma das classes, ao ser avaliado com todas as sequências do conjunto provavelmente terá seu desempenho aumentado. Se o conjunto de sequências for representativo o suficiente para todas as classes de programas existentes, então, para um novo programa a ser compilado todas as boas sequências podem ser avaliadas.

A estratégia de Purini e Jain possui uma fase inicial que consiste em utilizar compilação iterativa para criar um conjunto de 290 sequências de otimizações para 66 *microbenchmarks*. Em seguida, três técnicas, um algoritmo chamado *10-melhores*, um algoritmo chamado *12-melhores* e um algoritmo de clusterização selecionam apenas as melhores das 290 sequências geradas.

O algoritmo *10-melhores* extrai das 290 sequências as 10 sequências que aumentam o desempenho, com relação a sequência padrão do compilador, para o maior número de *benchmarks*. No algoritmo *12-melhores*, o *10-melhores* é aplicado, as sequências escolhidas são removidas para gerar um novo conjunto de sequências e em seguida, as seis melhores sequências são selecionadas. Esse processo se repete mais uma vez e os dois grupos de seis sequências geram o conjunto com as 12 melhores.

O algoritmo de clusterização é o *Kmeans* e tem como entrada um conjunto de vetores, de modo que cada vetor, o qual representa uma sequência, tem o tamanho igual a quantidade de *benchmarks*. Cada posição do vetor possui um valor lógico que indica se aquela sequência fornece *speedup* para o *benchmark* naquela posição. Cada vetor representa um ponto no espaço multidimensional e dois pontos próximos indicam que duas sequências melhoraram o desempenho para quase os mesmos programas. Ao ser executado, o algoritmo de clusterização gera 10 *clusters* de sequências que melhoram o desempenho para 10 grupos (classes) de programas e de cada *cluster*, a sequência que cobre o maior número de *benchmarks* é extraída para formar o pequeno grupo de boas sequências.

A abordagem com o algoritmo de clusterização foi a que apresentou os melhores resultados. Para as suítes de *benchmarks* Mibench e Polybench foram obtidos desempenhos médios de 9.8 e 11.7, respectivamente. É importante uma observação de que uma das suítes usadas para avaliação, o Polybench, por possuírem simples *kernels* de aplicações que executam rapidamente, foi empregada no conjunto de treino e não no de teste do trabalho atual, o qual avaliou o SPEC2006 e o cBench.

O trabalho de Purini e Jain se diferencia em vários aspectos do trabalho proposto nesta dissertação. Em primeiro lugar, pelo conteúdo submetido ao algoritmo de clusterização: enquanto a estratégia de Purini e Jain realiza um agrupamento de sequências, neste trabalho se agrupam programas. No algoritmo dos autores o objetivo da clusterização é descobrir um grupo de sequências que cobre o maior número de programas, ao passo que a estratégia deste trabalho consiste em descobrir grupos de programas que mais se assemelham ao programa de entrada.

A estratégia apresentada nesta dissertação aplica para o programa de entrada possivelmente um conjunto específico de sequências. Pois de todas as sequências provenientes do mecanismo de conhecimento prévio, somente aquelas associadas aos programas mais semelhantes são escolhidas para otimizar o programa de entrada. Ao passo que a abordagem de Purini e Jain aplica o mesmo conjunto de sequências a todos os programas submetidos à compilação.

Outros Métodos de Aprendizagem de Máquina

Long e O'Boyle (Long e O'Boyle, 2004) propuseram um método de aprendizagem baseado em instâncias para o PSF, o qual tenta aplicar a um novo programa de entrada otimizações associadas a programas similares compilados previamente. Cem transformações são aplicadas para cada programa do conjunto de treinamento e os valores de *speedup* são armazenados. Cada programa é caracterizado com informações estáticas do código-fonte, tais como, número de *arrays*, *loops* e operações aritméticas, que servem para medir a semelhança entre diferentes códigos. Um algoritmo de classificação separa os programas em classes, de modo que o programa de entrada é associado a uma delas. Todas as sequências dos programas da classe escolhida são submetidas a um algoritmo de k vizinhos mais próximos, o qual seleciona uma sequência para ser aplicada ao programa de entrada.

A avaliação experimental foi empregada com 16 programas das suítes de *microkernels* Livermore e Java Grande Forum. O *speedup* médio para a melhor sequência com relação ao programa não otimizado foi de 1.15.

Agakov et al. (Agakov et al., 2006) propuseram uma técnica independente de algoritmo de busca que foca a procura de sequências em áreas do espaço de busca em que a ocorrência de uma sequência de alto desempenho é mais provável. Em uma fase de treinamento inicial, programas são compilados com diversas sequências e seus *speedups* são armazenados. Posteriormente a técnica gera para cada programa um vetor de probabilidades, em que cada posição corresponde à probabilidade da otimização fornecer *speedup* ao programa. Essas probabilidades são dadas por meio de uma função da

quantidade de vezes que a otimização apareceu nas sequências da fase de treinamento. Dois modelos fornecem as probabilidades, um baseado em uma distribuição independente e o outro em uma distribuição de Markov.

Na fase de desenvolvimento, características estáticas do programa de entrada são extraídas e utilizadas para medir sua sua semelhança com os *benchmarks* de treinamento. Um algoritmo de classificação simples de vizinho mais próximo correlaciona o programa novo com algum já compilado da fase anterior e gera algumas sequências de acordo com as probabilidades das otimizações para compilar com o novo programa.

Com apenas duas avaliações na fase final, a abordagem de Agakov et al. obteve um *speedup* médio de 1.22 para o modelo de distribuição independente e 1.27 para o modelo de Markov. Os *benchmarks* avaliados foram a suíte de *microkernels* UTDSP.

Cavazos et. al. (Cavazos et al., 2007) propuseram o uso de *performance counters* para caracterizar programas. Os autores fizeram uso de um algoritmo de aprendizagem de máquina baseado em um modelo de regressão linear para escolher conjuntos de otimizações para serem aplicados ao programa de entrada. Cavazos et al. aplicaram sua estratégia para os *benchmarks* do SPEC2000 e obtiveram um desempenho similar àquele do *Combined Elimination* de Pan e Eigenmann (Pan e Eigenmann, 2006). No entanto, por utilizarem conhecimento prévio em um algoritmo de aprendizagem a quantidade de avaliações necessárias foi menor do que a dos outros autores.

Como Agakov et al., Malik (Malik, 2010) também selecionou programas semelhantes com aprendizagem de máquina, no entanto, a caracterização de cada programa é dada por informações do grafo de fluxo de dados. Essa estrutura intermediária do compilador é representada como um histograma, de modo que dois grafos isomórficos são representados pela mesma distribuição no histograma e dois grafos similares, mas não isomórficos, por distribuições semelhantes. Essas duas propriedades permitem computar a similaridade entre dois programas por meio da criação de vetores de características de informações dos histogramas. Em uma fase de treinamento, um algoritmo genético é usado para associar sequências aos programas de treinamento; em seguida, um modelo de aprendizagem seleciona quais as melhores otimizações a serem aplicadas resolvendo um problema de regressão. Árvore de decisão e máquinas de vetores de suporte (*SVM - Support Vector Machine*) foram as técnicas de aprendizagem avaliadas.

A caracterização de programas proposta por Malik proporcionou a seleção de sequências com desempenho ligeiramente superior ao da sequência fixa do compilador. Considerando a melhor configuração, para dez dos programas do Mibench, o ganho de *speedup* médio foi de 1.7% e 2.2% para a estratégia de árvore de decisão e SVM, respectivamente.

Park et al. (Park et al., 2011) avaliaram três seletores de sequências, que utilizam aprendizagem de máquina para mitigar o POF, de modo que um deles corresponde a um novo preditor baseado em torneios. Inicialmente a fase inicial é realizada associando sequências de alto desempenho aos programas, os quais são caracterizados por meio de *performance counters*. Todos os seletores possuem como entrada uma caracterização do programa a ser compilado.

O primeiro seletor tem como saída a probabilidade de cada otimização disponível pelo compilador aumentar o desempenho do programa. Essa probabilidade é obtida pela amostragem da melhor sequência de cada programa do conjunto de treinamento. Sequências são geradas para serem aplicadas ao programa de entrada por meio de escolha aleatória tendenciosa das otimizações baseada nas probabilidades. O segundo seletor tem como entrada, além da caracterização do programa, um vetor de *bits* que representa a habilitação ou não de cada otimização que forma uma sequência. Na fase de treinamento, o seletor analisa os *speedups* das sequências para fornecer como saída uma previsão do *speedup* fornecido pela sequência. O terceiro seletor possui duas sequências como entrada e sua saída é um valor lógico que indica se a primeira sequência é melhor do que a segunda. O treinamento também é realizado com informações das melhores sequências da fase inicial e a seleção é realizada por um algoritmo de competição que utiliza o seletor.

A avaliação empregada por Agakov et al. fez uso de alguns dos programas presentes nas coleções de *microkernels* UTDSP, NAS, Linpack, Polybench e Mibench. Considerando a melhor configuração, o *speedup* médio para as quatro primeiras suítes foi de 1.75. Ao passo que para as aplicações do Mibench o *speedup* médio para as sequências não vistas foi de 1.06.

Jantz e Kulkarni (Jantz e Kulkarni, 2013) realizaram um estudo sobre a viabilidade de estratégias para mitigar o PSF em um ambiente de compilação JIT (*Just-In-Time*). Dentre as contribuições dos autores, eles empregaram uma técnica de compilação iterativa (algoritmo genético) e outra de aprendizagem de máquina para selecionar otimizações em uma máquina virtual. Na fase de treinamento da segunda abordagem, Jantz e Kulkarni caracterizaram os programas com informações estáticas de código-fonte e selecionaram as otimizações para serem aplicadas ao programa de entrada com um algoritmo de regressão logística.

É interessante notar que a técnica de compilação iterativa obteve os melhores resultados, porém necessitou de um número maior de avaliações. Para duas suítes de *benchmarks*, SPECjvm98 e DaCapo, o AG teve ganho de 7% e 3.2%, respectivamente. Já a abordagem de aprendizagem de máquina apresentou ganho de 2.5% para o SPECjvm98 e desempenho equivalente à sequência padrão para o DaCapo.

A principal semelhança das estratégias de aprendizagem de máquina descritas com a abordagem de seleção de otimizações desta dissertação é a existência de algum mecanismo de conhecimento prévio de onde são coletadas informações que guiam a escolha das sequências de otimizações. Além disso, de maneira geral, esse mecanismo é construído similarmente: sequências são aplicadas a programas e algum tipo de informação sobre as melhores fases são armazenadas para posterior consulta. No entanto, diversas diferenças podem ser apontadas entre o este trabalho e os demais.

No trabalho desta dissertação, os programas são caracterizados com *performance counters* e apenas o trabalho de Park et al. (Park et al., 2011) também faz isso. Todos os outros trabalhos fazem uso de algoritmos de classificação, como regressão linear ou SVM, para escolher as sequências a serem aplicadas aos programas. Neste trabalho, um algoritmo de clusterização é empregado na seleção. Essa abordagem implica em uma característica interessante.

A maioria dos trabalhos citados, como o de Long e O'Boyle (Long e O'Boyle, 2004), Agakov et al. (Agakov et al., 2006), ou o de Malik (Malik, 2010), geram ou selecionam a melhor sequência considerando apenas o *benchmark* mais semelhante ao programa de entrada. No entanto, nem sempre a melhor sequência está associada ao programa mais similar. Como a abordagem de clusterização desta dissertação considera um grupo de *benchmarks*, um programa que não é o mais similar e que está associado a uma boa sequência tem a oportunidade de fornecê-la para ser aplicada ao programa de entrada.

O último aspecto a ser ressaltado é com relação aos *benchmarks*. Diferente da maioria das outras abordagens, neste trabalho os experimentos foram conduzidos com duas suítes de *benchmarks* relevantes no contexto de seleção de otimizações, o SPEC2006 e o cBench. Na seção que descreve a metodologia deste trabalho no Capítulo 4 as duas suítes são apresentadas.

2.4.3 Outras Estratégias

Chabbi et al. (Chabbi et al., 2011) propuseram um estudo experimental que possui algumas semelhanças com a estratégia de ordenação proposta nesta dissertação. Eles estudaram a relação de pares de otimizações em sequências. A pesquisa deles resultou em três técnicas que tentam reduzir o espaço de busca de sequências e prever o tempo de execução de um programa. No entanto, a que mais se assemelha com a estratégia proposta e que é descrita a seguir é a modelagem que os autores propuseram baseada em grafos.

A estratégia proposta por Chabbi et al. compila e executa um programa com todos os possíveis pares de otimizações em ordem e na ordem inversa. Por exemplo, se duas otimizações a e b pertencem ao conjunto de otimizações do compilador, então o programa é compilado com a sequência (a, b) e (b, a) e o resultado de desempenho com cada sequência é armazenado.

Os pares que obtiveram desempenho melhor do que o seu inverso são armazenados para compor o modelo e os pares que correspondem ao inverso são descartados. Um grafo é construído mapeando cada otimização como um nó e cada valor de desempenho de um par de otimizações como o peso da aresta que liga esses dois nós. Um algoritmo de ordenação topológica é executado para descobrir a ordem - a qual corresponde ao caminho mais curto do grafo - com que os nós (otimizações) estarão na sequência.

A proposta de Chabbi et al. se assemelha a deste trabalho por considerar o efeito de pares e otimizações e por fazer uso de um grafo para considerar o espaço de sequências, entretanto a maneira de modelar o problema é substancialmente diferente. O primeiro ponto a ser observado é que Chabbi et al. ponderam o peso das arestas do grafo com a informação de desempenho dos pares de otimizações para o programa de entrada. No trabalho desta dissertaçaõ, a abordagem de ordenação de sequência tenta tirar proveito de uma estrutura de conhecimento prévio para dar custo às arestas. Ela faz isso por considerar a frequência dos pares de otimizações das sequências provenientes desta estrutura para custear as arestas do grafo.

Essa diferença entre as duas estratégias possui uma consequência importante. Para criar o seu modelo de grafo, Chabbi et al. necessita executar o programa $2\binom{n}{2}$ vezes, valor que corresponde à quantidade de pares possíveis de serem formados com n otimizações. Ao passo que isso não é requerido na abordagem de ordenação proposta neste trabalho, já que as informações são provenientes do mecanismo de conhecimento prévio, o qual é gerado em uma fase *offline* do compilador.

Outra diferença importante é que no trabalho de Chabbi et al. os autores estão interessados principalmente em descobrir o menor caminho do grafo. Neste trabalho, o principal objetivo é descobrir pares de otimizações que sejam suficientemente significativos para impactar o aumento de desempenho fornecido pela sequência que eles pertencem. Isso significa que não necessariamente se busca o caminho de menor custo, mas sim o caminho que contenha os melhores pares de otimizações. Desse modo, um algoritmo heurístico aleatório e tendencioso é empregado para resolver o problema baseado em grafo que é criado. Essa abordagem busca induzir uma ordenação aleatória, porém que tende a considerar as informações do mecanismo de conhecimento prévio para descobrir bons pares de otimizações.

O trabalho de Chabbi et al. foi um estudo empírico sobre a influência de pares de otimizações no desempenho de sequências, de modo que não há resultados de *speedup*. Para realizar seus experimentos, os autores utilizaram alguns pequenos programas escritos em FORTRAN.

2.5 Considerações Gerais

O objetivo deste capítulo foi introduzir os conceitos relacionados à otimização em compiladores, apresentar os principais tipos de estratégia para solucionar o problema e retratar o estado da arte dos trabalhos atuais.

Para isso foram apresentados o PSF e POF, bem como as técnicas de compilação iterativa e aprendizagem de máquina. Em seguida, alguns dos principais trabalhos para otimização de programas foram apresentados.

Na Seção 2.4 foi ressaltado em diversos momentos a importância do conjunto de *benchmarks* empregados nos experimentos. Isso se deve ao fato de que para realizar uma avaliação experimental consistente e relevante, a coleção de programas deve ser suficientemente adequada e significativa para que os resultados possam ser estendidos à sistemas reais. Neste trabalho, as duas suítes avaliadas são relevantes no contexto de otimização em compiladores, principalmente o SPEC2006.

O próximo capítulo apresentará as duas abordagens propostas para mitigar o PSF e o POF e como elas são aplicadas de maneira integrada para solucionar os dois problemas.

3

Estratégias de Seleção e Ordenação de Fase

Este capítulo apresenta as estratégias propostas para a mitigação do PSF e POF, bem como a forma como elas foram organizadas no *framework* de otimização de programas proposto.

A Seção 3.1 apresenta a arquitetura do *framework* e a Seção 3.1.1 descreve a implementação do mecanismo de conhecimento prévio utilizado pelas duas estratégias. A Seção 3.2 detalha a estratégia de seleção de sequências, descrevendo o módulo do *framework* responsável pela seleção e as etapas envolvidas: coleta de características, seleção de agrupamentos, clusterizador e seleção de sequências. Por último, a Seção 3.3 apresenta o módulo que realiza a ordenação de sequências, descrevendo o algoritmo de transformação do POF e o algoritmo de resolução baseado em colônia de formigas.

3.1 Visão Geral

O *Framework* de Seleção e Ordenação de Fase (FSOF) implementa as estratégias de seleção e ordenação de fase propostas neste trabalho. Ele é composto por três componentes: Módulo de Seleção de Fase (MSF), Módulo de Ordenação de Fase (MOF) e Espaço Exploratório de Sequências (ES - *Exploratory Space*). O MSF e o MOF encapsulam um compilador otimizante a fim de compilar programas com sequências escolhidas de ES e sequências geradas a partir de informações de ES. O ES é uma estrutura composta por *benchmarks*, que representam o espaço de programas, e sequências de otimizações que fornecem *speedups* a eles, servindo como base de conhecimento para as duas estratégias.

O MSF realiza a seleção de uma sequência por meio de uma abordagem de aprendizagem de máquina baseada em clusterização, a qual realiza o agrupamento de programas do espaço exploratório com o programa de entrada. Esse agrupamento considera a semelhança entre os programas, os quais são caracterizados por meio de *performance counters*. As melhores sequências associadas a cada programa do ES, que foi agrupado no mesmo grupo do programa de entrada, é aplicada ao novo programa para aumentar seu desempenho.

No MOF, um algoritmo é proposto para transformar o problema da ordenação em um problema baseado em grafo. Essa transformação é realizada por um algoritmo que, dada uma sequência de entrada, tenta inferir o efeito de aplicar suas otimizações em outra ordem, baseado em informações das sequências presentes no espaço exploratório. O principal objetivo da estratégia é tentar descobrir pares de otimizações da sequência de entrada que possam beneficiar o desempenho do programa.

O problema baseado em grafo que é gerado pelo algoritmo de transformação é resolvido por meio de um algoritmo de colônia de formigas (Dorigo et al., 1996) e a solução é mapeada de volta ao problema original.

A Figura 3.1 apresenta a arquitetura do FSOF. Inicialmente, um programa de entrada p_x é submetido ao MSF, o qual aciona um algoritmo de clusterização que é executado com p_x e os *benchmarks* do conjunto de programas presentes em ES. Dentro do MSF, um algoritmo gera um grupo de programas g_K que possui p_x como elemento. As melhores sequências associadas aos *benchmarks* de ES pertencentes a g_K são aplicadas ao programa p_x e a sequência s que fornece o maior desempenho, mais o grupo g_K são encaminhados ao módulo de ordenação.

Dentro do MOF um algoritmo de transformação, a partir da sequência não ordenada s, transforma o POF em um Problema do Caixeiro Viajante Assimétrico (PCVA). Um algoritmo de colônia de formigas é então executado para fornecer soluções para o PCVA. Cada solução é mapeada para uma sequência de otimizações que é avaliada e aquela sequência s' que apresenta o melhor desempenho é fornecida como saída do sistema, juntamente com a versão otimizada de p_x.

Os detalhes de cada componente do FSOF são descritos nas próximas seções. A Seção 3.1.1 descreve como é o processo de construção do espaço exploratório ES. A Seção 3.2 apresenta o MSF e a estratégia de seleção e a Seção 3.3 apresenta o MOF e a estratégia ordenação.

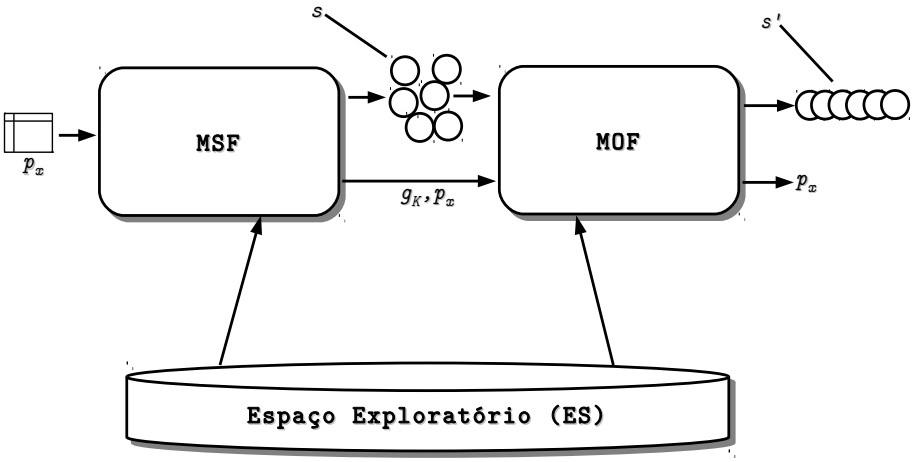


Figura 3.1: Arquitetura do FSOF.

3.1.1 O Espaço Exploratório de Sequências

Os dois módulos do FSOF fazem uso de um mecanismo de conhecimento prévio chamado *espaço exploratório de sequências*, ou espaço exploratório. No MSF o espaço exploratório é consultado com o objetivo de buscar sequências para um novo programa, já no MOF esta estrutura é vasculhada na tentativa de descobrir informações relevantes com relação a ordem das sequências.

O espaço exploratório é uma estrutura em que programas são associados a subconjuntos de sequências que fornecem *speedup* a eles. Em notação matemática, considere um conjunto de programas $P = \{p_1, p_2, p_3, ..., p_m\}$ e o conjunto de otimizações $O = \{o_1, o_2, o_3, ..., o_n\}$, o espaço exploratório é uma função que mapeia um programa de P para um par ordenado em que o primeiro elemento corresponde a uma sequência de otimizações e o segundo um valor real como descrito a seguir

$$ES\colon P \rightarrow (\wp(O) \times \Re) | ES(p) = (s, r) \Rightarrow r > 1 \tag{3.1}$$

onde $p \in P$ é um programa, $s \in \wp(O)$ é uma sequência de otimizações e $r \in \Re$ é um valor real que representa o *speedup* alcançado pela sequência s quando compilada com p. Note que na Expressão 3.1 a restrição de que $r > 1$ indica que as sequências pertencentes a ES devem fornecer *speedup* maior do que 1 para o programa p ao qual elas estão associadas.

A expressão 3.1 também indica que a ordem com que as otimizações são aplicadas ao programa p não é relevante, pois a sequência s é um subconjunto não ordenado de $\wp(O)$. Na construção do ES, para refletir o fato de que uma sequência $s \in \wp(O)$ não

possui ordem definida, as otimizações que formam todas as sequências são escolhidas aleatoriamente.

Pelo fato do ES ser uma função, a notação $ES(p)$ retorna o conjunto de todas as sequências de ES e seus respectivos valores de *speedup* associadas ao programa p.

Uma forma de interpretar o espaço exploratório é como um vetor de listas indexado pelo nome dos programas. Cada posição do vetor corresponde a uma lista de pares ordenados em que o primeiro elemento representa uma sequência e o segundo elemento o seu valor de *speedup* quando a sequência é aplicada ao programa. A Tabela 3.1 apresenta um exemplo de espaço exploratório com sequências de tamanho 3.

Programa	Sequências
p_1	$< \{o_1, o_5, o_4\}, 1.200 >, < \{o_4, o_3, o_5\}, 1.180 >, < \{o_3, o_9, o_1\}, 1.300 >, ...$
p_2	$< \{o_2, o_1, o_3\}, 1.020 >, < \{o_2, o_4, o_1\}, 1.014 >, < \{o_7, o_2, o_3\}, 1.010 >, ...$
p_3	$< \{o_4, o_2, o_1\}, 1.001 >, < \{o_1, o_4, o_3\}, 1.230 >, < \{o_5, o_2, o_5\}, 1.020 >, ...$
...	...

Tabela 3.1: Exemplo de espaço exploratório com sequências de tamanho 3.

Note que é possível fazer uso da notação de função para selecionar todas as sequências e seus valores de *speedup* do espaço exploratório associadas a algum programa, por exemplo $ES(p_1) = \{< \{o_1, o_5, o_4\}, 1.200 >, < \{o_4, o_3, o_5\}, 1.180 >, < \{o_3, o_9, o_1\}, 1.300 >, ...\}$ retorna todas as sequências e *speedups* associados com o programa p_1.

Construção de ES

O algoritmo para construção do espaço exploratório é descrito nesta seção. Para computar os valores de *speedup*, cada programa de ES foi compilado com uma sequência $s_{baseline}$, a qual corresponde a habilitação de uma das sequências fixas da infraestrutura de compilação utilizada O0, O1, O2 e O3. Para decidir qual delas escolher, uma avaliação experimental com todos os *benchmarks* empregados nos experimentos foi conduzida, de modo que para a maioria deles a sequência O2 obteve o melhor desempenho, assim foi definido que $s_{baseline} = O2$. O valor de *speedup* de uma sequência s aplicada a um programa p foi computado pela expressão

$$SPEEDUP(p, s) = \frac{T(p, s_{baseline})}{T(p, s)} \tag{3.2}$$

onde a função $T(p, s)$ retorna o tempo de execução do programa p compilado com a sequência s.

Ao verificar a documentação das transformações fornecidas pela LLVM (Lattner e Adve, 2004), algumas diretrizes são fornecidas de quais otimizações podem preceder ou suceder uma otimização para aumentar sua efetividade. Baseado nessas informações, o Algoritmo 1 foi aplicado a cada sequência. Como exemplo, o manual sugere que sempre ao aplicar *sccp* (*Sparce Conditional Constant Propagation*) é uma boa ideia aplicar *dce* (*Dead Code Elimination*) logo em seguida. Dessa maneira, sempre que *sccp* está presente em uma sequência o algoritmo insere *dce* na posição seguinte. As versões do programa geradas pela sequência original e a versão modificada são avaliadas e apenas a que obtém melhor desempenho é armazenada. Esse algoritmo simples é semelhante ao sugerido por Purini e et al. (Purini e Jain, 2013).

Algoritmo 1 Melhoria de sequência

Entrada: s /* Sequência */
Saída: s' /* Sequência melhorada */
  $s' \leftarrow s$
  for $i = 0$ to $size(s')$ **do**
    if $s'[i]$ == -constprop **then**
      $s'[i+1] \leftarrow$ -die
    else if $s'[i]$ == -sccp **then**
      $s'[i+1] \leftarrow$ -dce
    else if $s'[i]$ == -scalarrepl **then**
      $s'[0] \leftarrow$ -scalarrepl
    else if $s'[i]$ == -scalarrepl-ssa **then**
      $s'[0] \leftarrow$ -scalarrepl-ssa
    else if $s'[i]$ == -inline **then**
      $s'[i-1] \leftarrow$ -inline-cost
    else if $s'[i]$ é a primeira otimização de *loop* **then**
      $s'[i-1] \leftarrow$ -loops
    else if $s'[i]$ == -pre-verify **then**
      $s'[n] \leftarrow$ pre-verify
    else if $s'[i]$ == -verify **then**
      $s'[n] \leftarrow$ -veriry
    end if
  end for

A construção do espaço exploratório foi realizada por meio de um algoritmo aleatório de compilação iterativa. Cada programa do conjunto de *benchmarks* foi compilado com 500 sequências, todas de tamanho fixo r. A seguir, uma descrição dos passos para a construção do espaço exploratório é fornecida:

- Para cada programa $p \in P$, compilar e executar p com $s_{baseline}$ e guardar o tempo de execução $T(p, s_{baseline})$.

- Para cada programa $p \in P$ gerar 500 sequências $\{s'_1, s'_2, ..., s'_{500}\}$ de tamanho fixo aleatoriamente e usar o Algoritmo 1 para gerar 500 sequências adicionais $\{s''_1, s''_s, ..., s''_{500}\}$.
  - Compilar e executar o programa p com cada sequência s'_i e s''_i, obter a sequência s_i, a qual é melhor entre s'_i e s''_i e obter os tempos de execução $\{T(p, s_1), T(p, s_2), ..., T(p, s_{500})\}$.
  - Calcular os *speedups* para cada sequência s_i e guardar somente aquelas sequências e seus *speedups* em que $SPEEDUP(p, s_i) > 1$.

3.2 O Módulo de Seleção de Fase

O MSF tenta otimizar um programa de entrada compilando-o com sequências que estão associadas a alguns dos *benchmarks* de ES. Esses *benchmarks* são escolhidos de acordo com a semelhança deles com o programa que se deseja otimizar. A avaliação da semelhança entre os programas é realizada por um algoritmo de clusterização que tenta identificar o grupo (*cluster*) de *benchmarks* que possui características mais similares ao programa de entrada.

O pressuposto para essa estratégia é o mesmo de diversos outros trabalhos de aprendizagem de máquina (Agakov et al., 2006; Cavazos et al., 2007; Cavazos e O'Boyle, 2006; Jantz e Kulkarni, 2013; Long e O'Boyle, 2004; Malik, 2010; Park et al., 2011; Thomson et al., 2010), o qual consiste em compilar um programa não visto com sequências que aumentaram o desempenho de programas semelhantes a ele.

A grande maioria das estratégias que utilizam aprendizagem de máquina tenta identificar apenas um único programa mais semelhante ao programa de entrada para fornecer sequências para aumentar o desempenho. Com isso se considera que na amostra do espaço de programas presente na fase de treinamento do compilador exista um programa significativamente semelhante ao programa de entrada. No entanto, devido ao espaço de programas ser infinitamente grande, dificilmente um programa terá esse nível de similaridade, o que pode diminuir a efetividade de uma estratégia que escolhe um único programa semelhante.

Para caracterizar os programas, cada um deles é representado por vetores de características que formam pontos em um espaço multidimensional. O algoritmo de clusterização atua nesse espaço tentado agrupar pontos que estejam próximos, ao mesmo tempo que tenta identificar grupos com a maior distância possível entre si.

A Figura 3.2 ilustra um agrupamento de programas. Por conveniência, neste exemplo cada programa é representado por apenas duas características: *acessos a memória* e

instruções aritméticas. Cada vez que um novo programa é submetido à compilação, o MSF executa o algoritmo de clusterização com os demais programas de ES. O algoritmo criou três grupos para nove exemplos iniciais e o novo programa é atribuído àquele grupo em que ele está mais próximo.

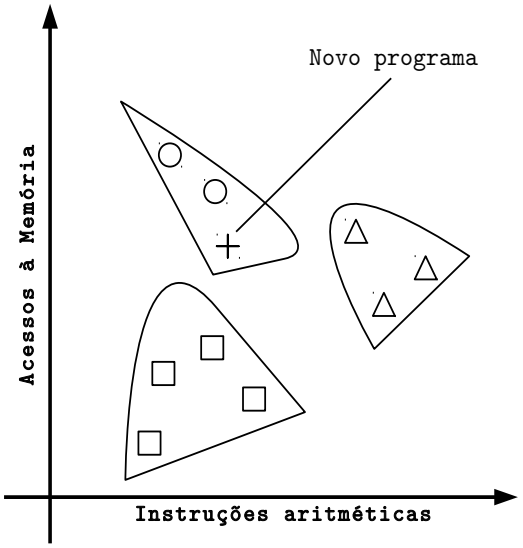


Figura 3.2: Ilustração de algoritmo de clusterização para agrupamento de programas.

Para gerar os vetores de característica é necessário obter informações que retratem o comportamento do programa. Uma alternativa seria caracterizá-lo com informações estáticas do código, no entanto, essa abordagem não representa o comportamento durante o tempo de execução (Cavazos et al., 2007). No MSF, cada programa é caracterizado por meio de contadores de desempenho de hardware (*performance counters*), de modo que cada elemento dos vetores de características possui um valor de contador. Apesar dessa abordagem necessitar de pelo menos uma execução da aplicação, ela possui o benefício de retratar o comportamento dinâmico do programa.

A Figura 3.3 apresenta a arquitetura do MSF. O primeiro componente é o Coletor de Características, que recebe como entrada o conjunto de m *benchmarks* $P = \{p_1, p_2, p_3, ..., p_m\}$ que estão em ES, mais o programa de entrada p_x e retorna um conjunto de vetores de características $U = \{u_1, u_2, u_3, ..., u_m, u_x\}$, em que cada elemento $u_i = \{c_{i1}, c_{i2}, c_{i3}, ..., c_{i\beta}\}$ corresponde ao vetor de características do programa i, u_x é o vetor de características de p_x, c_{ij} é o valor do *performance counter* j do programa i e β é a quantidade de *performance counters*.

O Seletor de Agrupamentos recebe o conjunto U e chama o Clusterizador passando como parâmetro um valor k que representa a quantidade de *clusters* a serem criados. A quantidade de vezes que o Seletor de Agrupamentos chama o Clusterizador é condicionada

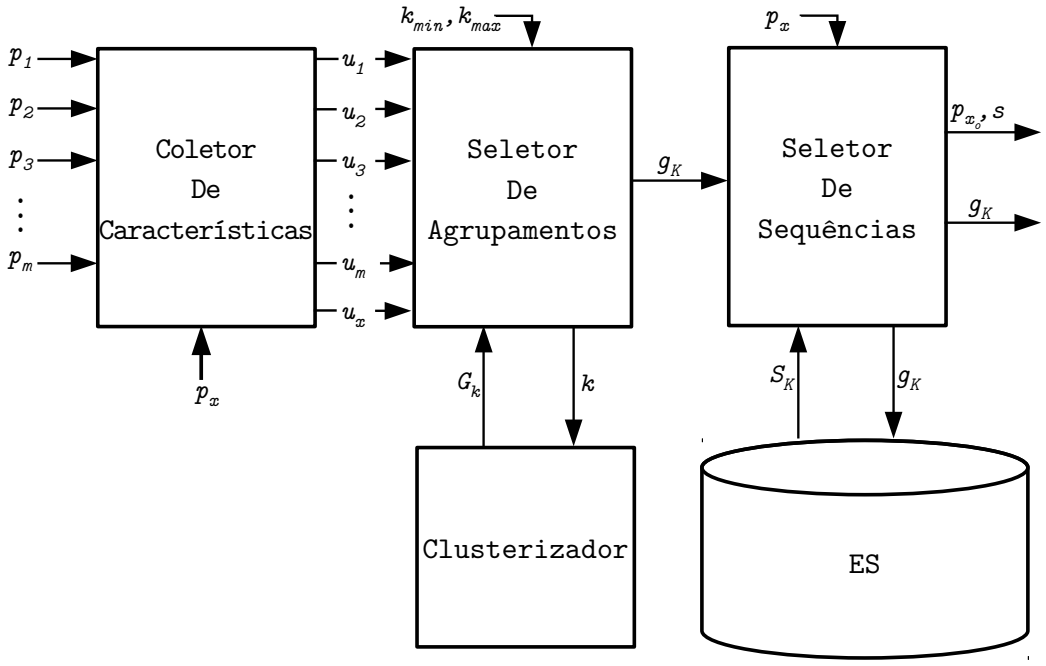


Figura 3.3: Arquitetura do Módulo de Seleção de Fase

pelos parâmetros de entrada k_{min} e k_{max}, de modo que $k_{min} \leq k \leq k_{max}$. A cada chamada o Clusterizador retorna um agrupamento $G_k = \{g_1, g_2, g_3, ..., g_k\}$ contendo k *clusters*, de modo que cada *cluster* corresponde a um subconjunto de programas de $P \cup \{p_x\}$. Após obter todos os agrupamentos $G_{k_{min}}$ até $G_{k_{max}}$, o Seletor de Agrupamentos avalia cada um deles por meio de uma função de qualidade de *cluster* $M(G)$. O grupo de programas g_K, que pertence ao agrupamento que obtém o menor valor de M, é então submetido ao Seletor de Sequências que seleciona de ES o conjunto S_K, que contém a melhor sequência associada a cada programa de g_K. Por fim, o Seletor de Sequências compila e executa p_x com cada uma das sequências selecionadas e fornece o executável otimizado que obteve o melhor desempenho p_{x_o}, a sequência s utilizada para compilá-lo e o grupo g_K.

É importante notar que o MSF pode ser executado isoladamente dentro do FSOF - sem a necessidade de ordenar a sequência com o MOF. Neste caso, apenas p_{x_o} e s são necessários como saída. O grupo de programas g_K é utilizado para a integração com o MOF.

As seções seguintes apresentam cada um dos componentes do MSF detalhadamente.

3.2.1 Coleta de Características

A primeira tarefa do MSF consiste em coletar as características dos programas, a qual é realizada pelo Coletor de Características. Esse processo de coleta é caracterizado

pela compilação e execução de cada programa p_i de ES e o programa de entrada p_x, com nenhuma otimização habilitada. Desse modo, os valores dos *performance counters* refletem o real comportamento dinâmico dos programas.

Em seguida, os *performance counters* coletados para cada programa i são mapeados para um vetor de características u_i e cada elemento de u_i é normalizado pelo total de instruções do programa (TOT_INS). Dessa forma, é possível generalizar as estatísticas computadas para todos os *benchmarks* independentemente do tempo que ele executa.

Note que a quantidade de *benchmarks* (pontos no espaço multidimensional) em cada experimento é um pouco menor do que a de *performance counters*. Essa situação significa que a quantidade de amostras é inferior ao total de dimensões, embora tipicamente o inverso seja desejado, pois a densidade de um espaço de n dimensões requer muitas vezes essa quantidade de amostras para que os pontos não fiquem esparsos excessivamente. Essa situação, a qual é conhecida como problema da dimensionalidade, pode ser solucionada aumentando a quantidade de amostras ou reduzindo a quantidade de dimensões do espaço. A primeira alternativa seria ideal, porém devido ao tempo necessário para executar a quantidade de programas requeridos tornaria a estratégia inviável. A segunda alternativa tem a desvantagem de possivelmente se perder informação, porém é possível reduzir esta perda para níveis aceitáveis. Logo, optou-se pela segunda opção e a técnica utilizada para redução da dimensionalidade foi a Análise de Componentes Principais (ACP) (Jackson, 1991).

Essa técnica utilizada na estatística identifica padrões nos dados expressando-os de uma maneira que realça suas diferenças e similaridades. Ela faz isso por procurar vetores ortogonais n-dimensionais, chamados componentes principais, que melhor ajustam os dados, de modo que podem ser projetados em um espaço muito menor por descartar alguns dos vetores. A ACP é descrita detalhadamente em (Jackson, 1991) e sua implementação disponível na ferramenta de aprendizagem de máquina Weka (WEKA, 2013) foi utilizada.

O Coletor de Características aplica a ACP no espaço formado pelos vetores de $U = \{u_1, u_2, u_3, ..., u_m, u_x\}$ e utiliza apenas os α componentes principais de cada u_i que cobrem 99% da amostra. Esses componentes principais formam o novo vetor de característica $U = \{u_1, u_2, u_3, ..., u_m, u_x\}$, em que cada vetor $u_i = \{c_{i1}, c_{i2}, c_{i3}, ..., c_{i\alpha}\}$, de modo que $\alpha < \beta$.

É importante ressaltar que a coleta de *performance counters* para os programas de P ocorre apenas uma vez durante a instalação do compilador, de modo que não faz parte do tempo de compilação de p_x. Já os contadores para p_x são coletados para cada nova compilação.

3.2.2 Clusterização

O algoritmo de clusterização possui como um de seus parâmetros um valor k que corresponde a quantidade de *clusters* a serem criados. Como essa informação não é conhecida, o Seletor de Agrupamentos experimenta diversos valores de k variando entre k_{min} e k_{max} até encontrar um agrupamento que melhor retrata o espaço. Para decidir qual é o melhor agrupamento dentre todos os criados, o Seletor de Agrupamentos emprega a medida de qualidade de agrupamento sugerida por Siddheswar (Siddheswar Ray, 1999). Essa medida considera dois aspectos do agrupamento: a proporção da variância dos pontos dentro de cada *cluster* (distância intra-cluster) e a proporção da variância dos centros de *clusters* (distância inter-cluster).

Essa medida leva em conta o objetivo do método de clusterização que é criar *clusters* compactos e ao mesmo tempo esparsos no espaço com relação aos outros *clusters*. Um *cluster* compacto significa que as distâncias entre os seus pontos e o centro do *cluster* (centroide) é pequena e *clusters* esparsos significam que a distância entre os centroides de cada *cluster* são tão maiores quanto o possível.

Para determinar o quanto um *cluster* é compacto, a medida de Siddheswar computa a distância quadrática entre cada ponto de um *cluster* para o seu centro. Para computar em um único valor a proximidade de todos os pontos com o centroide, Siddheswar calcula a média do valor das distâncias entre todos os pontos e seus centros de *cluster*. A essa medida é dado o nome de distância intra-cluster e ela é definida pela expressão

$$M_{intra} = \frac{1}{m} \sum_{i=1}^{k} \sum_{x \in C_i} |x - z_i|^2 \tag{3.3}$$

onde m é a quantidade de pontos, k é o número de *clusters*, z_i é o centroide do *cluster* C_i e $|a-b|$ é a função módulo ou norma entre dois vetores a e b em um espaço multidimensional, de modo que a expressão $|x - z_i|^2$ calcula a distância quadrática entre um ponto x pertencente ao *cluster* C_i e o seu centro de *cluster* z_i. Os dois somatórios da Expressão 3.3 computam a soma das distâncias entre todos os m pontos pertencentes ao espaço e seus centros de *cluster* z_i, a qual é dividida pela quantidade total de pontos. A medida M_{intra} corresponde então ao valor médio da distância quadrática entre todos os m pontos no espaço e seus centros de *cluster*. M_{intra} tende a zero quanto mais próximos os pontos estiverem dos seus centroides, de modo que agrupamentos de *clusters* compactos tendem a minimizar M_{intra}.

Além de *clusters* compactos, um dos objetivos da clusterização é identificar *clusters* que tenham seus centros mais distantes possíveis no espaço. Desse modo, Siddheswar propõe a distância inter-cluster M_{inter}, que corresponde ao valor mínimo das distâncias quadráticas entre todos os centroides. Quanto maior o valor desta distância, mais distantes estão os *clusters* uns dos outros, de modo que o objetivo é maximizá-la. M_{inter} é expressa como

$$M_{inter}= \min(|z_i - z_j|^2), 1 \leq i \leq k-1 \tag{3.4}$$

$$i+1 \leq j \leq k. \tag{3.5}$$

onde z_i e z_j são os centroides i e j, respectivamente, e k é a quantidade de *clusters*. Essa distância se deseja maximizar, pois a distância entre os dois *cluster* mais próximos deve ser a maior possível. A fim de determinar a qualidade de um agrupamento se almeja minimizar M_{intra} e maximizar M_{inter}, de modo que a combinação das duas medidas é a medida de qualidade de um agrupamento $M(G)$ que é dada pela expressão

$$M(G)= \frac{M_{intra}}{M_{inter}}. \tag{3.6}$$

Quanto menor o valor M, maior é a qualidade do *cluster*. Cada vez que o Seletor de Agrupamentos chama o Clusterizador passando um valor k, este retorna um agrupamento $G_k = \{g_1, g_2, ..., g_k\}$, onde cada g_i é um *cluster* de programas. Com as informações dos *clusters* de G_k o Seletor de Agrupamentos calcula $M(G_k)$ para todos os valores de k até que se encontre o agrupamento G_k^* que obtém o menor valor de M e é o agrupamento escolhido. O grupo de programas $g_K \in G_k^*$ que possui o programa p_x é submetido ao Seletor de Sequências.

A seguir, os dois algoritmos empregados pelo Clusterizador para realizar os agrupamentos são apresentados.

Algoritmos de Clusterização

No MSF a clusterização dos programas foi realizada por meio de dois algoritmos, o Kmeans e o EM (*Expectation Maximization*). Foi utilizada a implementação destes algoritmos disponível na ferramenta de aprendizagem de máquina Weka (WEKA, 2013).

O objetivo é avaliar a influência de diferentes algoritmos nos resultados obtidos, principalmente com relação à escolha dos programas de ES que compõem o grupo do programa de entrada. A seguir é fornecida uma descrição dos dois algoritmos.

Kmeans O Kmeans (Hartigan, 1975) é um clássico algoritmo de aprendizagem de máquina para realizar clusterização, o qual tenta minimizar a distância quadrática de cada ponto do *cluster* com relação ao seu centro. Inicialmente, k pontos são escolhidos aleatoriamente como centros de *cluster* e em seguida um processo iterativo de 3 passos se inicia:

1. Para cada centroide um novo grupo é criado e baseado na métrica de distância euclidiana, cada ponto é atribuído ao grupo que possui o centro de *cluster* mais próximo.

2. Os centroides de cada *cluster* tem seus valores atualizados para o valor médio entre as distâncias de todos os pontos do *cluster*.

3. Os passos 1 e 2 são repetidos com os novos centroides e o processo continua até que entre duas iterações os mesmos pontos sejam atribuídos aos mesmos grupos, momento em que o algoritmo estabiliza.

Um problema intrínseco ao Kmeans é que o parâmetro k deve ser fornecido antecipadamente para o algoritmo. No entanto, não é possível ao MSF saber qual é este valor *apriori*, já que esse é justamente o conhecimento que se deseja obter. Assim, a Seção 3.2.2 apresentou uma técnica para determinar a quantidade de *clusters*. Outra característica do Kmeans é que ele pode não estabilizar, de modo que um máximo de iterações é imposto como limite para encontrar os *clusters*.

EM (Expectation-Maximization) O algoritmo EM (Dempster et al., 1977) é um método de clusterização probabilístico que consiste de dois passos: calcular as probabilidades de *clusters* e calcular os parâmetros da distribuição dos dados de entrada. Esses parâmetros são os valores das médias e desvios padrão da distribuição dos dados para cada *cluster*. O objetivo do algoritmo é descobrir qual é essa distribuição. O EM começa com suposições iniciais para esses parâmetros, usa esses valores para calcular as probabilidades de cada instância e usa essas probabilidades para reestimar os parâmetros em um processo iterativo. Os passos do algoritmo EM são brevemente descritos a seguir:

1. Atribuir um valor aleatório para os valores de média e desvio padrão de cada *cluster* e estimar a distribuição de cada *cluster*.

2. Passo *Expectation*: Calcular a probabilidade de cada instância estar em um *cluster*, baseado nas distribuições de cada *cluster*.

3. Passo *Maximization*: Utilizar as probabilidades do passo anterior e repetir o passo reestimando os valores de média e desvio padrão de cada *cluster*. O processo se encerra quando um limiar de erro é alcançado.

Diferente do Kmeans, o EM é um algoritmo em que cada instância possui uma probabilidade de estar em um *cluster*, de modo que ela será atribuída ao *cluster* que possui maior probabilidade. O EM também necessita que a quantidade de *clusters* inicial seja fornecida, desse modo, da mesma maneira que o Kmeans, a técnica descrita na Seção 3.2.2 é empregada pelo MSF para descobrir qual o melhor valor de k.

3.2.3 Seleção de Sequências

Após determinar qual o melhor conjunto de *clusters*, o Seletor de Agrupamentos fornece g_K para o Seletor de Sequências. O Seletor de Sequências escolhe a melhor sequência associada a cada programa de ES que pertence a g_K para compilar p_x.

Para avaliar o desempenho, o Seletor de Sequências compila e executa p_x com todas as sequências selecionadas e fornece o executável que obtém o melhor desempenho p_{x_o} e a sequência s correspondente.

3.3 O Módulo de Ordenação de Fase

O MOF é o módulo do FSOF responsável por ordenar a sequência de entrada e fornecer uma versão do programa de entrada compilada com esta fase. Esse módulo realiza o processo de ordenação tentando aproveitar o conhecimento da ordem das sequências de ES que estão associadas aos programas mais semelhantes ao programa de entrada.

Esse conhecimento é obtido por meio da análise de pares de otimizações das sequências de ES. O objetivo da estratégia é encontrar padrões de comportamento de pares de otimizações que sejam significativos para a sequência de entrada. É baseado no comportamento desses pares que o MOF modifica a ordem de aplicação das otimizações da sequência.

O comportamento de um par de otimizações para o MOF é considerado como um custo. Por exemplo, se após analisar o espaço exploratório, o módulo considera que aplicar uma otimização o_i antes de o_j diminui o desempenho de um programa, ele dá um custo, possivelmente alto, ao par (o_i, o_j) que reflita esse comportamento. O módulo

fornece um custo a todos pares formados com as n otimizações da sequência de entrada e em seguida, tenta juntar todos os pares para formar a sequência final que será utilizada para otimizar o programa.

Para realizar a atribuição dos custos aos pares de otimizações, o MOF computa a frequência dos pares. Dada uma sequência de entrada s, o módulo consulta em ES a frequência com que todos os possíveis pares de serem formados com as otimizações de s ocorrem nas sequências associadas aos programas do espaço exploratório. Como as sequências de ES conseguem fornecer *speedup* aos *benchmarks*, o módulo é guiado pela suposição de que pares com uma alta frequência são significativos para aumentar o desempenho do programa de entrada.

Para ilustrar o conceito de frequência no MOF, a Figura 3.4 apresenta um exemplo do módulo de ordenação. Nesta figura, existem quatro otimizações (`a, b, c` e `d`), de modo que a primeira coluna apresenta todos os pares que envolvem a otimização `a`. A segunda coluna contém a frequência com que aquele par ocorre nas sequências de ES e a terceira, a quantidade de vezes que as duas otimizações ocorrem na sequência (inclusive na ordem inversa).

O MOF dá o custo para cada par de otimização inversamente proporcional à frequência do par com relação ao total. Deste modo, no exemplo da Figura 3.4, o par `(a, d)` terá o menor custo, pois do total de 11 vezes que as otimizações `a` e `d` aparecem nas sequências de ES, 10 sequências contém `(a, d)` e apenas uma `(d, a)`. Por consequência, o par `(d, a)` terá o maior custo, o que indica ao MOF que aplicar a otimização `a` antes de `d` é melhor do que o inverso. Com relação aos demais pares não é possível ao MOF deduzir a ordem em aplicá-los, pois o custo de um par com relação ao seu inverso é muito semelhante. No entanto, o objetivo da abordagem é descobrir principalmente aqueles pares mais significativos. Além disso, se deseja que a geração das soluções seja realizada de maneira aleatória, porém tendenciosa a escolher sempre os pares de menor custo.

Se for considerado que as otimizações são vértices, o problema de juntar os pares é muito parecido com o bem conhecido PCVA, em que tem-se um conjunto de n vértices, existe um custo de ir de cada vértice para a outra e o objetivo é encontrar o circuito de menor custo. De fato, o MOF utiliza um algoritmo para transformar uma versão do POF em uma do PCVA. É importante perceber que a versão assimétrica do Problema do Caixeiro Viajante é mais adequada do que a original, pois o custo de aplicar uma otimização o_i antes de uma o_j é diferente de o_j antes de o_i.

Após transformar uma instância do problema da ordenação em uma do PCVA, este último é solucionado por um algoritmo *Ant System*. Essa heurística foi escolhida para resolver o problema, pois ela considera a distância entre os pares de otimizações na geração

Par de otim.	Freq.	Total
(a, b)	4	8
(a, c)	5	8
(a, d)	10	11
(b, a)	4	8
(c, a)	3	8
(d, a)	1	11

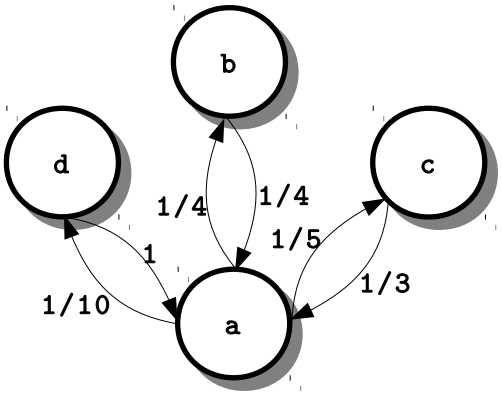


Figura 3.4: Ilustração de como o MOF trata os pares de otimizações.

das soluções. Uma solução é construída por uma formiga percorrendo cada nó e escolhendo o próximo baseado em uma função da distância com o próximo nó. Na Figura 3.4, se uma formiga está em `a`, é possível controlar a probabilidade dela escolher o nó com maior frequência por um ajuste de parâmetro e assim , na maioria das vezes, ela terá maiores chances de escolher o nó `d` do que os demais. A seguir os algoritmos de transformação do POF para o PCVA e *Ant System* são apresentados.

3.3.1 Transformação do POF para o PCVA

Antes de apresentar como ocorre a transformação do POF para o PCVA, os dois problemas serão descritos de maneira a realçar suas semelhanças. Sobre o POF, será apresentada uma reformulação do problema original, que é baseada na questão de pares de otimizações.

Inicialmente a formulação do PCVA como um problema baseado em grafo. Dado um grafo $G' = (V', E', c')$, em que $V' = \{v'_1, v'_2, v'_3, ..., v'_n\}$ é o conjunto de vértices, $E' = (V' \times V') - \{(v', v') \mid v' \in V'\}$ é o conjunto de arestas e $c' : E \to Q_{\geq 0}$ é uma função de custo que atribui a cada aresta um valor não negativo e admite-se a existência da regra $c'(i, j) \neq c'(j, i)$. O PCVA é o problema de encontrar um ciclo Halmitoniano mínimo em G'.

A nova formulação do POF é como segue. Dado uma tupla $G = (O, E, c, p_x)$ onde O é um conjunto de otimizações, E é um conjunto de pares de otimizações $E = (O \times O) - \{(o, o) \mid o \in O\}$, o qual $(o_i, o_j) \in E$ significa a aplicação do par de otimizações (o_i, o_j) ao programa p_x nesta ordem, e $c : E \to Q_{\geq 0}$ é uma função do custo de aplicar um par de otimizações $c(o_i, o_j)$ ao programa p_x. O POF é o problema de encontrar uma permutação das otimizações de O de maneira que a soma dos custos entre as otimizações seja mínima.

Dada as definições dos dois problemas acima, é possível estabelecer uma correspondência entre os elementos de G' e G: o conjunto O pode ser mapeado para o conjunto V', o conjunto E para E' e a função c para c'. Dessa forma, o processo de transformação consiste em dois passos: mapear otimizações do POF para vértices do PCVA e mapear o custo de aplicar pares de otimizações do POF para custos de arestas do PCVA. O primeiro passo é feito simplesmente mapeando cada otimização do POF para um vértice do PCVA diretamente como a seguir:

$$\forall o_i \in O_{\wedge} v'_i \in V' \Rightarrow o_i = v'_i, \;\; i = 1, 2, ..., n \tag{3.7}$$

O segundo passo consiste em atribuir pesos as arestas do PCVA. Para isso é necessário definir qual o custo em aplicar um par de otimizações (o_i, o_j), nesta ordem, no programa p_x. Como mencionado anteriormente este custo é computado por uma função inversamente proporcional à frequência com que uma otimização o_i aparece antes de outra o_j em ES.

Para auxiliar na computação deste custo define-se a função $Freq\_Prog$, a qual dada duas otimizações o_i e o_j e um programa p_k pertencente a ES, retorna a frequência com que o_i aparece antes de o_j nas sequências associadas ao programa p_k

$$Freq\_Prog(p_k, o_i, o_j) = |\, \{s \in Dom(ES(p_k)) \,|\, (o_{i \wedge} o_j \in s) \;_{\wedge}\; o_i \prec o_j\} \,| \tag{3.8}$$

onde o operador $\prec$ indica que a otimização o_i precede a otimização o_j na sequência s e o operador $|x|$ retorna o tamanho do conjunto x. O conjunto do lado direito da Expressão 3.8 contém todas as sequências de ES que estão associadas ao programa p_k e que possuem a otimização o_i precedendo o_j, de modo que $Freq\_Prog$ corresponde ao tamanho deste conjunto. Como ES é constituído por diversos programas, o custo de um par $c(o_i, o_j)$ poderia ser a soma de $Freq\_Prog$ para todos os programas de ES: $\sum_{p_k \in P} Freq\_Prog(p_k, o_i, o_j)$. No entanto este cálculo do custo segue a propriedade de que apenas programas similares a p_x devem ser analisados.

Se o cálculo do custo c for realizado analisando as sequências de todos os programas P pertencentes a ES provavelmente ocorrerá um problema de que sequências associadas a programas não similares ao programa de entrada p_x também serão analisadas. Desta forma, um par de otimizações com um alto valor de frequência para programas muito diferentes do programa de entrada, não seria representativo com relação ao próprio p_x. Ao invés de computar a frequência para todos os programas, o MOF realiza essa computação

apenas para aqueles que são mais semelhantes ao programa de entrada. A identificação desta classe de programas similares ao programa de entrada p_x é feita pelo MSF. Antes de iniciar o algoritmo de transformação o MSF é acionado para gerar e fornecer ao MOF o grupo de programas g_K (descrito na Seção 3.2), o qual contém o programa p_x. Com o conhecimento dos programas que são similares a p_x, o MOF calcula a frequência de pares de otimizações apenas para as sequências de ES associadas ao grupo em que p_x pertence.

Desse modo, a função $Freq$, que retorna a frequência com que duas otimizações aparecem nas sequências associadas aos programas de g_K de ES, é definida como

$$Freq(o_i, o_j) = \sum_{p_k \in g_K} Freq\_Prog(p_k, o_i, o_j) \tag{3.9}$$

$Freq(o_i, o_j)$ é a soma da quantidade de vezes (sequências) que as otimizações o_i e o_j aparecem nessa ordem nas sequências de cada programa de g_K.

Se a frequência $Freq(o_i, o_j) < Freq(o_j, o_i)$, o MOF considera que a aplicação de (o_j, o_i) é uma melhor escolha do que na ordem contrária. Isso significa que quanto maior o valor de $Freq(o_j, o_i)$ (o que implica um menor valor de $Freq(o_i, o_j)$), maior deve ser o custo de aplicar (o_i, o_j) a p_x. Assim, o custo de aplicar um par (o_i, o_j) é dado pela frequência do inverso desse par $c(o_i, o_j) = Freq(o_j, o_i)$.

Nem todos os pares de otimizações aparecem em todas as sequências de ES, logo pode ocorrer uma alta variação para dois pares diferentes sobre os seus valores de frequência, assim o custo para cada par é normalizado da seguinte maneira

$$c(o_i, o_j) = \frac{Freq(o_j, o_i)}{Freq(o_i, o_j) + Freq(o_j, o_i)} \tag{3.10}$$

em que a soma no denominador da fração corresponde a quantidade de vezes total que as otimizações o_i e o_j apareceram nas sequências dos programas de g_K. Com esta normalização o custo $c(o_i, o_j)$ sempre ficará entre 0 e 1, de modo que, por exemplo, se $c(o_i, o_j) = 0.2$ então $c(o_j, o_i) = 0.8$. Note que se haverem ocorrências apenas do par (o_j, o_i) (e por consequência nenhuma ocorrência de (o_i, o_j)), $c(o_i, o_j) = 1$, o valor de custo mais alto possível.

Com a função de custo definida, a transformação do POF para o PCVA é realizada por mapear o custo de cada par de otimizações do POF para arestas do PCVA. O Algoritmo 2 resume como é realizado todo o processo de transformação. O algoritmo recebe como entrada o conjunto de otimizações O, ES, o programa p_x e o grupo de programas g_K

e retorna um grafo G' do PCVA. Para calcular a frequência o algoritmo utiliza uma matriz chamada $freq$ em que cada entrada $freq[o_i][o_j]$ representa a frequência com que o_i aparece antes de o_j. Esta matriz é preenchida percorrendo todas as sequências associadas aos programas similares a p_x e cada vez que uma otimização o_i aparece antes de uma o_j, $freq[o_i][o_j]$ é incrementado. Note que a utilização desta matriz faz com que cada sequência de ES necessite ser visitada apenas uma vez. Em seguida, o algoritmo atribui os valores da função de custo c e caso o par (o_i, o_j) não ocorra em nenhuma sequência $(freq[o_i][o_j] + freq[o_j][o_i] = 0)$, o custo para ele é igual a 1. Por último, o algoritmo mapeia cada otimização de O para um vértice de V', cada aresta de E para uma aresta de E' e a função c para c'.

3.3.2 Resolução do PCVA

O algoritmo a ser utilizado para solucionar o PCVA é o *Ant System* proposto por Dorigo (Dorigo et al., 1996). O *Ant system* é uma heurística para resolver problemas de otimização combinatória baseado no comportamento coletivo de formigas artificiais. Cada formiga é responsável por criar uma solução e a cada passo do algoritmo ela deixa uma informação de qualidade daquela solução (feromônio) para que as próximas formigas possam considerar na construção das soluções seguintes. O feromônio é o meio de comunicação coletivo pelo qual as formigas decidem qual o melhor caminho escolher.

O processo de construção de solução de uma formiga é realizado por deslocá-la de nó em nó probabilisticamente, de maneira que ela considere a informação do feromônio e a distância entre cada nó. Ao final da geração de uma solução, cada formiga deposita em cada aresta uma quantidade de feromônio que é uma função da distância do tamanho do *tour*.

Ao final do algoritmo um vetor $melhorTour$ é fornecido, o qual contém o melhor *tour* entre todas as soluções geradas pelas formigas. É importante notar que cada posição deste vetor corresponde a uma otimização de O e a ordem dos nós que ele descreve, corresponde à ordem de aplicação das otimizações.

O Algoritmo 3 retrata a implementação do *Ant System* neste trabalho. Neste algoritmo o grafo $G' = (V', E', c')$ do PCVA é representado pela matriz de adjacência $n \times n$ $pcva$, na qual cada linha e coluna representam o conjunto de cidades e o valor de cada entrada $pcva[i][j]$ representa o custo de ir de uma cidade i para uma cidade j dado pela função c'.

Inicialmente a matriz de feromônio $feromonio[n][n]$ é iniciada com um valor constante b. Cada entrada $feromonio[i][j]$ desta matriz corresponde ao feromônio deixado pelas formigas na aresta $pcva[i][j]$. Após a inicialização da trilha de feromônio o algoritmo

Algoritmo 2 Algoritmo de transformação do POF para o PCVA

Entrada: $O = \{o_1, o_2, o_3, ..., o_n\}$ /* Sequência s (não ordenada) de entrada */
g_K /* Grupo de programas similares */
ES /* Espaço Exploratório */
p_x /* Programa de entrada */
Saída: $G' = (V', E', c')$ /* PCVA */
$G = (O, E = O \times O - \{(o, o) \,|\, o \in O\}, c, p_x)$ /* POF */
$freq[o_i][o_j] \leftarrow 0, \forall o_{i \wedge} o_j \in V$
for each $p \in Dom(ES)$ **do**
  for each $s \in Dom(ES(p)) \,|\, p \in g_K$ **do**
    if $o_{i \wedge} o_j \in s \wedge o_i \prec o_j \; \forall i, j = \{1, 2, 3, ..., n\}$ **then**
      $freq[o_i][o_j] \leftarrow freq[o_i][o_j] + 1$
    end if
  end for
end for
for $o_i \in O$ **do**
  for $o_j \in O$ **do**
    if $freq[o_i][o_j] + freq[o_j][o_i] > 0$ **then**
      $c(o_i, o_j) \leftarrow \frac{freq[o_j][o_i]}{freq[o_i][o_j] + freq[o_j][o_i]}$
    else
      $c(o_i, o_j) \leftarrow 1$
    end if
  end for
end for
for $o_i \in O$ **do**
  $v'_i \leftarrow o_i$
  $V' \leftarrow V' \cup \{v'_i\}$
end for
for $(o_i, o_j) \in E$ **do**
  $E' \leftarrow E' \cup \{(o_i, o_j)\}$
  $c'(o_i, o_j) \leftarrow c(o_i, o_j)$
end for

entra em um processo iterativo até que algum critério de parada seja satisfeito, nesta implementação o algoritmo para após completar 100 ciclos.

A cada iteração, primeiramente, cada uma das formigas de *colonia*$[n]$ constrói uma solução. A função `Escolher_Cidade` determina qual será a próxima cidade da formiga: se a formiga está iniciando seu *tour*, quando $it = 1$, esta função escolhe uma cidade aleatoriamente para posicioná-la; se $it > 1$ `Escolher_Cidade` utiliza uma função de probabilidade p_{ij}^k, a qual corresponde a probabilidade da formiga k ir da cidade i para a cidade j.

Algoritmo 3 *Ant System* para o PCVA

```
Entrada: pcva[n][n]      /* Matriz de adjacência n × n do PCVA */
  m                      /* Quantidade de formigas */
  α, β                   /* Importância do feromônio e visibilidade */
Saída: melhorTour[n]     /* Melhor tour encontrado pelas formigas */
  colonia[m]             /* Vetor de estruturas: colonia[k].tour, colonia[k].tamTour */
  feromonio[i][j] ← b, ∀ij ∈ 1..n
  while critério não-satisfeito do
    for it ← 1 to n do
      for k ← 1 to m do
        Escolher_Cidade(colonia[k].tour, it);
      end for
    end for
    for k ← 1 to m do
      colonia[k].tamTour ← Calcular_Tamanho_Tour (colonia[k].tour);
    end for
    melhorTour ← Atualizar_Melhor_Tour (colonia, melhorTour)
    for i ← 1 to n do
      for j ← 1 to n do
        feromonio[i][j] ← feromonio[i][j] * ρ * Σ_{k=1}^{m} Δτ_ij^k
      end for
    end for
    for k ← 1 to m do
      Apagar_Memoria (colonia[k]);
    end for
  end while
```

Após construir as soluções para toda colônia, o tamanho de *tour* de cada i-formiga $colonia[i].tamTour$ é atualizado pela função `Calcular_Tamanho_Tour`. Em seguida, o melhor *tour* gerado até este ponto $melhorTour$ é atualizado pela função `Atualizar_Melhor_Tour`.

O próximo passo consiste em atualizar a trilha de feromônio. O parâmetro ρ representa a taxa de evaporação do feromônio e a quantidade do resíduo $\Delta\tau_{ij}^{k}$ deixado por cada formiga k na aresta (i, j) é dado pela função

$$\Delta\tau_{ij}^{k} = \frac{Q}{T_k} \tag{3.11}$$

onde Q é um parâmetro empírico, constante e T_k é o tamanho do *tour* da formiga k ($colonia[k].tamTour$). Em seguida, o *tour* e seu tamanho é apagado da memória de cada formiga com a função `Apagar_Memoria` e o processo se repete.

Como mencionado anteriormente, cada formiga escolhe a próxima cidade baseada em uma função de probabilidade do feromônio e do tamanho do *tour*. Essa probabilidade, utilizada pela função `Escolhe_Cidade`, de uma formiga k posicionada em um nó i escolher j como o próximo nó é dada pela expressão

$$p_{ij}^{k} = \begin{cases} \frac{[\tau_{ij}]^{\alpha}[\eta_i]^{\beta}}{\sum_{k \notin Visitados_k} [\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}} & \text{se} \quad j \notin Visitados_k \\ 0 & \text{caso contrário} \end{cases} \tag{3.12}$$

onde τ_{ij} é a quantidade de feromônio na aresta (i, j), $\eta_{ij} = 1/d_{ij}$ é a visibilidade do vértice j pela formiga a posicionada em i, d_{ij} é a distância entre i e j, $Visitados_k$ é o conjunto de nós visitados pela formiga, α é um parâmetro de importância de feromônio e β é a importância da visibilidade da formiga.

É importante observar que quanto maior o valor de β, mais uma formiga tende a escolher o nó seguinte que está mais próximo, pois é o que possui menor valor de visibilidade. Por outro lado, altos valores de α fazem com que as formigas escolham arestas que tiveram maior tráfego anteriormente. O parâmetro β é especialmente importante, pois o fato das formigas escolherem arestas com custo mais baixo está relacionado com a proposta de construção de sequências para o POF, a qual consiste em tentar escolher pares de otimizações de menor custo. Na Seção 4.4.2 do Capítulo 4, é realizada uma discussão a respeito dessa influência da configuração do algoritmo com relação às sequências geradas.

Após obter o melhor *tour* por meio do Algoritmo 3, o MOF faz o mapeamento de cada posição i de $melhorTour$ para as otimizações correspondentes de O, gerando uma sequência ordenada pelo melhor *tour* encontrado pelas formigas. O MOF executa o *Ant System* L vezes para obter as sequências que serão utilizadas para otimizar o programa e após gerar todas as soluções, ele avalia todas as L sequências e fornece aquela sequência s', juntamente com a versão do programa compilado com ela, que obteve melhor desempenho.

3.4 Considerações Gerais

Este capítulo apresentou a estrutura do FSOF, o mecanismo de conhecimento prévio denominado *ES* e as duas estratégias de otimização de programas implementadas nos módulos MSF e MOF. O próximo capítulo apresenta uma avaliação experimental do espaço exploratório, das estratégias de seleção e ordenação separadamente e do FSOF como um todo.

4

Avaliação Experimental

O principal objetivo deste capítulo é avaliar o desempenho das estratégias de seleção e ordenação, bem como o desempenho geral do FSOF. Os parâmetros a serem avaliados são:

- *Speedup* alcançado pela melhor sequência para o MSF, MOF e FSOF.
- Número de avaliações realizadas com o MSF, MOF e FSOF.

O capítulo é organizado com a Seção 4.1 descrevendo todos os materiais e métodos empregados na realização dos experimentos. A Seção 4.2 apresenta uma avaliação de *speedup* das sequências do espaço exploratório. A Seção 4.3 apresenta os resultados obtidos com o MSF e uma avaliação comparativa entre os dois algoritmos de clusterização empregados. Por fim, a Seção 4.4 apresenta os resultados obtidos com o MOF e uma avaliação do algoritmo de transformação e de colônia de formigas.

É importante ressaltar que os resultados do MSF foram comparados com a estratégia de Thomson et al. (Thomson et al., 2010) e os resultados gerais do FSOF foram comparados com o trabalho de Pan e Eigenmann (Pan e Eigenmann, 2006).

4.1 Metodologia

Plataforma Os experimentos deste trabalho foram conduzidos com a infraestrutura de compilação LLVM versão 3.4 (Lattner e Adve, 2004). O FSOF e a LLVM executaram sob três máquinas idênticas baseadas na arquitetura Intel x86_64. Os três computadores possuíam processadores Core I7-2600 executando em uma frequência de 3.4GHz com cache L1 de instruções e dados de 32KB, cache L2 de 256KB, cache L3 de 8MB e memória RAM

de 4GB. O sistema operacional executando nas máquinas foi o Ubuntu 11, sob a versão de *kernel* 2.6.32-5-amd64.

Performance Counters Os 43 *performance counters* para caracterização dos programas disponíveis nas máquinas, separados por classes de instruções, são descritos na Tabela 4.1. A coleta deles foi realizada por meio das ferramentas de *profile* de aplicação PAPI versão 5.1.0.2 (Mucci et al., 1999), e PerfSuite versão 1.1.2 (Kufrin, 2005). O Perfsuite fornece um utilitário chamado `psrun`, o qual realiza a instrumentação do arquivo executável com instruções da biblioteca do PAPI. O PAPI fornece um conjunto de funções e estruturas de dados que permitem obter os valores dos *performance counters* disponíveis na máquina.

Classe de Instruções	**Performance Counters**
Saltos	BR_CN, BR_TKN, BR_NTK, BR_MSP, BR_PRC
Cache L1	L1_DCM, L1_ICM, L1_TCM, L1_LDM, L1_STM
Cache L2	L2_DCM, L2_ICM, L2_TCM, L2_STM, L2_DCH, L2_DCA, L2_DCR, L2_DCW, L2_ICA, L2_TCW, L2_ICH, L2_ICR, L2_TCA, L2_TCR
Cache L3	L3_TCM, L3_DCA, L3_DCR, L3_DCW, L3_ICA, L3_ICR, L3_TCA, L3_TCR, L3_TCW
TLB	TLB_DM, TLB_IM
Ciclos	STL_ICY, TOT_CYC, REF_CYC
Instruções	TOT_INS, LD_INS, SR_INS, BR_INS
Ponto Flutuante	FDV_INS

Tabela 4.1: Conjunto de performance counters disponíveis nas máquinas Intel(R) Core I7-2600 3.4GHz.

Benchmarks Três suítes de *benchmarks* foram empregadas nos experimentos: Polybench (Polybench., 2013), SPEC2006 (Henning, 2006) e o Collective Benchmark (cBench) (cBench., 2013). As duas primeiras suítes de *benchmarks* são compostas por programas científicos e aplicações de alto desempenho. O Polybench corresponde a um conjunto de *kernels* de aplicações e o SPEC2006 é uma suíte de programas de ponto inteiro e flutuante de uso intensivo de processador para estressar os sistemas de processamento e memória. O cBench é uma suíte de *benchmarks* baseada no Mibench (Guthaus et al., 2001), a qual é composta por programas da área de sistemas embarcados. Todos os programas do Polybench e do cBench são escritos em linguagem C e os do SPEC2006 são aqueles implementados em C e C++. Todos os *benchmarks* das suítes correspondem a programas

sequenciais. Devido a erros de compilação, o *benchmark* `483.xalancbmk`, do SPEC2006, e o `office_ispell`, do cBench não foram utilizados.

Os *benchmarks* foram divididos em um grupo de treino e um grupo de teste. O conjunto de treino é composto pelos programas do espaço exploratório e o conjunto de teste pelos programas que serão otimizados. Os programas do Polybench compõem o conjunto de treino e os do SPEC2006 e do cBench fazem parte do conjunto e teste. Os programas do Polybench foram escolhidos para compor ES, pois eles correspondem à *kernels* de aplicações com tempo de execução rápido, o que é desejável na construção do mecanismo de conhecimento prévio.

Todas as entradas dos *benchmarks* do Polybench e do cBench foram configuradas com o valor padrão. O SPEC2006 possui três tipos de conjunto de dados, `test`, `train` e `ref`, dos quais o `test` possui as entradas de menor tamanho e o `ref` as de maior tamanho. Devido ao alto tempo de execução das entradas do conjunto `ref`, que impossibilitariam a avaliação da suíte, as entradas do conjunto `train` foram empregadas nos experimentos.

Otimizações Todos os resultados de desempenho foram gerados com relação a uma das sequências padrão da LLVM. Para descobrir qual das sequências padrão era a melhor, todos os *benchmarks* foram executados cada um dos níveis da LLVM (O1, O2 e O3) e o melhor foi escolhido como referencial. Para a maioria dos *benchmarks*, O2 obteve o melhor desempenho e ele foi considerado a sequência *baseline*.

Para geração das sequências, as otimizações foram divididas em um grupo de 45 e outro de 70 otimizações. O primeiro grupo contém todas as 45 diferentes otimizações presentes na sequência O2 e um outro é composto por todas as otimizações de O2, mais 35 otimizações. A Tabela 4.2 apresenta as otimizações que compõem os dois grupos.

Parâmetros de desempenho Como o principal objetivo era avaliar o desempenho das sequências de otimizações com relação ao *baseline*, dois parâmetros foram analisados:

Speedup Razão entre o tempo de execução do programa p compilado com a sequência *baseline* e com uma sequência s dado na Seção 3.1.1, a qual é repetida aqui:

$$SPEEDUP(p,s) = \frac{T(p, s_{baseline})}{T(p,s)} \tag{4.1}$$

Ganho

$$Ganho(p,s) = SPEEDUP(p,s) - 1 \times 100\% \tag{4.2}$$

Otimizações			
O2	-adce	-basicaa	-basiccg
	-constmerge	-correlated-propagation	-deadargelim
	-domtree	-dse	-early-cse
	-functionattrs	-globaldce	-globalopt
	-gvn	-indvars	-inline
	-inline-cost	-instcombine	-ipsccp
	-jump-threading	lazy-value-info	-lcssa
	-licm	-loop-deletion	-loop-idiom
	-loops	-loop-rotate	-loop-simplify
	-loop-unroll	-loop-unswitch	-memcpyopt
	-memdep	-no-aa	-notti
	-preverify	-prune-eh	-reassociate
	-scalar-evolution	-sccp	-simplifycfg
	-sroa	-strip-dead-prototypes	-tailcallelim
	-targetlibinfo	-tbaa	-verify
Outras	-always-inline	-argpromotion	-bb-vectorize
	-block-placement	-break-crit-edges	-constprop
	-dce	-die	-instsimplify
	-ipconstprop	-loop-extract	-loop-extract-single
	-loop-instsimplify	-loop-reduce	-lower-expect
	-loweratomic	-lowerinvoke	-lowerswitch
	-mem2reg	-mergefunc	-mergereturn
	-partial-inliner	-scalarrepl	-scalarrepl-ssa
	-sink		

Tabela 4.2: Conjunto de otimizações empregadas nos experimentos.

4.1.1 O Workflow Experimental

O processo de compilação de cada arquivo-fonte dos *benchmarks* representa o ciclo básico de realização de todos os experimentos. Esse processo foi realizado com o auxílio da infraestrutura de compilação LLVM (Lattner e Adve, 2004), a qual é formada por uma coleção de ferramentas de compiladores e bibliotecas reusáveis que permitem otimizar o código-fonte de aplicações. Essa infraestrutura foi escolhida, pois fornece uma grande variedade de otimizações para otimizar os programas, além de ser um compilador *mainstream* na atualidade.

A LLVM fornece uma representação intermediária (RI) do código-fonte chamada representação SSA. Todas as otimizações da Tabela 4.2 são aplicadas na RI SSA de um programa. Isso implica que o processo de otimização de um programa é independente da arquitetura da máquina-alvo. A RI de um arquivo-fonte é manipulada por três ferramentas

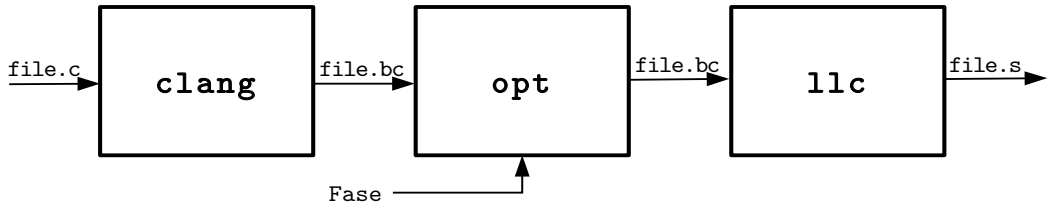


Figura 4.1: Ciclo de compilação de um arquivo com a infraestrutura LLVM.

da LLVM: o compilador *clang*, o otimizador *opt* e o compilador estático *llc*. A Figura 4.1 retrata o ciclo de compilação de um arquivo com a LLVM.

O arquivo-fonte escrito em C ou C++ é submetido ao *clang*, o qual é configurado para gerar o código na RI da LLVM com a flag -O0 habilitada. Isso garante que nenhuma otimização seja habilitada. O *opt* recebe este arquivo *bitcode* (`.bc`) e o transforma aplicando uma sequência de otimizações fornecida pelo FSOF. O *llc* recebe a RI da LLVM otimizada pelo *opt* e gera o código de montagem com instruções da arquitetura alvo. Este arquivo `.s` é submetido então ao linker *as* e *ld*, o qual gera o executável para a máquina-alvo.

Durante as avaliações cada execução foi realizada de maneira sequencial, com apenas um núcleo executando em um instante de tempo. Para evitar a sobrecarga imposta pelo sistema operacional, todos os serviços foram desligados, exceto aqueles essenciais para o seu funcionamento e para a funcionalidade de acesso remoto das máquinas.

A ferramenta *time* foi utilizada para realizar a coleta dos tempos de execução. Os experimentos foram conduzidos nas máquinas com as tecnologias de mudança dinâmica de frequência do processador para proteção térmica e otimização de desempenho desabilitadas para não ocorrer influência nos resultados obtidos pelo *time*. Cada execução foi repetida 10 vezes para garantir um mínimo de confiabilidade estatística e o valor médio entre todas as rodadas foi empregado. No caso de coleta de *performance counters*, o valor de cada contador corresponde a média obtida para as 10 execuções.

4.1.2 Configuração Experimental

Configuração *ES* O principal objetivo do espaço exploratório é fornecer sequências de alto desempenho associadas a programas representativos, para as fases de seleção e ordenação do FSOF. Desse modo, a qualidade das sequências escolhidas afetam a efetividade de *ES*.

Um dos principais fatores que podem afetar a qualidade das sequências é a quantidade de otimizações disponíveis. Sequências geradas com poucas otimizações podem não

oferecer muitas oportunidades de se otimizar o código e sequências com muitas otimizações podem gerar quantidade excessiva de modificações que, ao invés de melhora, podem causar degradação do desempenho do programa. Outros trabalhos que fizeram uso de estruturas semelhantes a *ES* geraram sequências com as seguintes quantidades de otimizações disponíveis, 38, 46, 20, 82, 121, 60, 88, 86, 45 e 29 (Agakov et al., 2006; Cavazos et al., 2007; Cavazos e O'Boyle, 2006; Haneda et al., 2005; Hoste e Eeckhout, 2008; Kisuki et al., 2000b; Leather et al., 2009; Lokuciejewski et al., 2011; Park et al., 2011; Thomson et al., 2010). A fim de avaliar a influência desse parâmetro, este trabalho gerou sequências para *ES* a partir de conjuntos de 45 e 70 otimizações, quantidades que são próximas ao valor médio (61,5) dos trabalhos dos outros autores. Do primeiro conjunto, são geradas sequências de tamanho 40 e as otimizações são pertencentes ao nível -O2 da LLVM, o qual corresponde à sequência *baseline*. Do segundo conjunto, sequências de tamanho 50 são geradas e cada fase é composta pelas 45 otimizações do primeiro conjunto, mais 35 transformações e análises.

A geração de sequências para *ES* apenas com as otimizações dos níveis de otimização da LLVM almeja tornar os resultados mais justos do ponto de vista de uma avaliação comparativa, já que o cálculo dos *speedups* foram realizados com relação ao nível O2 da LLVM. Por outro lado, o espaço exploratório com sequências possuindo mais otimizações visa simular um ambiente mais realístico, em que uma grande quantidade de otimizações são aplicadas para aumentar o desempenho do programa.

A Tabela 4.3 resume as versões de espaço exploratório geradas. Note que são escolhidas apenas 500 sequências de um total 45^{40} e 70^{50} possíveis para `ES.45` e `ES.70`, respectivamente.

Nome	**# Otimizações**	**# Sequência**	**Tam. Seq.**	**Fonte**
`ES.45`	45	500	40	O2
`ES.70`	70	500	50	O2 + Outras

Tabela 4.3: Configurações do espaço exploratório.

Configuração MSF A Tabela 4.4 apresenta os parâmetros de configuração para a estratégia de seleção. Ao parâmetro K_{min} foi atribuído o valor 2, como sendo a quantidade mínima de *clusters*. O parâmetro K_{max} foi configurado igual a 14 para que cada *cluster* tivesse a oportunidade de ter pelo menos dois dos 29 programas de *ES*. O algoritmo de clusterização foi executado 10 vezes para cada configuração e o melhor agrupamento foi escolhido. Todas as quatro combinações de algoritmos (`Kmeans, EM`) e espaço exploratório (`ES.45, ES.70`) foram avaliadas.

Parâmetros
$K_{min} = 2$
$K_{max} = 14$
$\#\ tentativas = 10$
$Algoritmos = \{$ `Kmeans, EM` $\}$
$ES = \{$ `ES.45, ES.70` $\}$

Tabela 4.4: Parâmetros de configuração do MSF.

Configuração MOF A configuração do MOF pode ser visualizada na Tabela 4.5. $L = 10$ sequências foram geradas com o algoritmo e a melhor foi escolhida para aplicar ao programa em cada execução. A quantidade de formigas é igual ao número de otimizações (n). Os outros parâmetros do algoritmo de colônia de formigas $Ciclos$, α, β, ρ, Q e c foram configurados com os valores padrão retratados na literatura (Dorigo et al., 1996). No entanto, na Seção 4.4.2, a fim de avaliar a influência da mudança do parâmetro β, que representa a visibilidade da formiga, alguns experimentos foram conduzidos com $\beta = \{1, 5, 30\}$.

É importante notar que o MOF foi executado, com relação a avaliação de desempenho, para todas as configurações do MSF.

Parâmetros
$L = 10$
$m = n$
$Ciclos = 100$
$\alpha = 1$
$\beta = 5$
$\rho = 0.99$
$Q = 100$
$c = 100$

Tabela 4.5: Parâmetros de configuração do MOF.

4.2 Avaliação do Espaço Exploratório

Esta seção apresenta os *speedups* das sequências presentes em `ES.45` e `ES.70`. Para esse fim, o comportamento da distribuição dos *speedups* para cada programa foram avaliados por meio de gráficos de violino (Hintze e Nelson, 1998). Esta ferramenta estatística é uma junção do *boxplot* com um estimador de densidade de *kernel*, que permite visualizar a distribuição dos dados. Se for traçada uma reta vertical no centro de um violino, as duas

metades são espelhos umas das outras e correspondem à distribuição da amostra. Essa representação da distribuição é conseguida com um estimador de *kernel* e a duplicação dela torna a visualização mais eficiente. No interior de cada violino a reta vertical no centro indica o intervalo em que está presente o primeiro, segundo e terceiro quartil do *boxplot* e o ponto representa a mediana.

A Figura 4.2 apresenta a distribuição de *speedups* para o espaço exploratório `ES.45`. É possível notar que todos os programas possuem ao menos uma sequência que fornece *speedup*, o que garante que qualquer um deles é capaz de fornecer sequências para serem usadas nas estratégias de seleção e ordenação. Os speedups dos programas são caracterizados por distribuições unimodais (`correlation, covariance, bicg, cholesky, gesummv, symm, durbin, gramschmidt, ludcmp, fdtd2d e jacobi2dimper`), bimodais (`2mm, 3mm, doitgen, gemm, gemver, mvt, syr2k, syrk, trisolv, trmm, dynprog, lu, floydwarshall, reg_detect, adi, fdtdapml e seidel2d`) ou trimodais (`atax`). Para a maioria dos *benchmarks* o valor da mediana das distribuições dos programas foi menor do que 1.0 e o valor de *speedup* para a melhor sequência foi menor do que 1.10. Os programas em que uma grande porção das sequências forneceram *speedup* maior do que 1.10 foram `gemm` e `dynprog`.

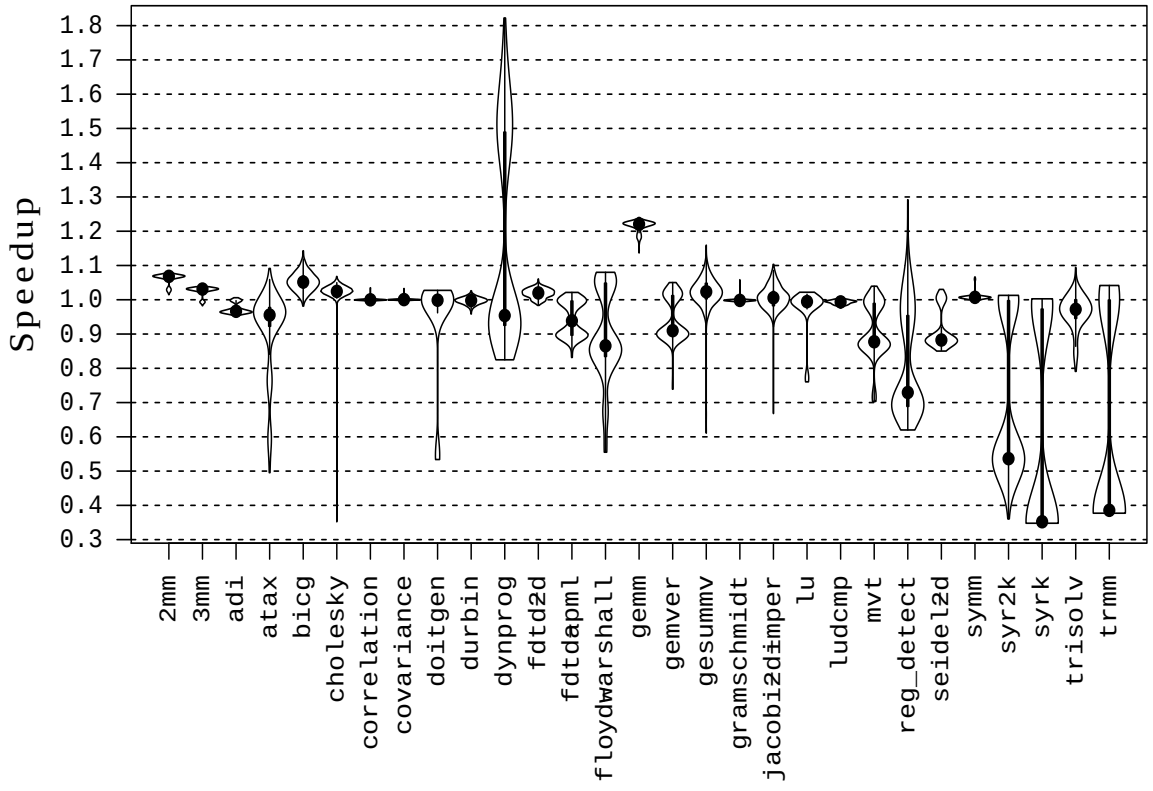


Figura 4.2: Distribuição de speedups para o espaço exploratório `ES.45`

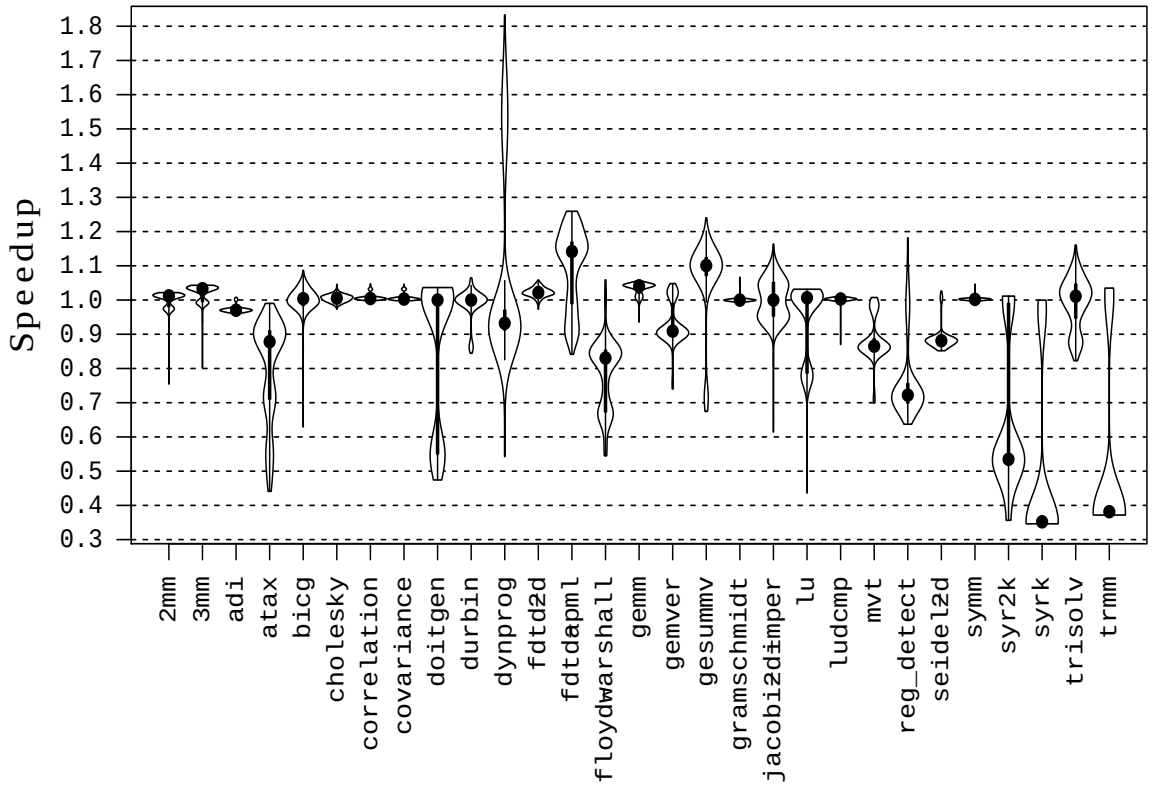


Figura 4.3: Distribuição de speedups para o espaço exploratório `ES.70`

A Figura 4.3 apresenta as distribuições dos *speedups* das sequências de `ES.70`. O primeiro aspecto a ser observado neste gráfico é a sua semelhança com o gráfico da Figura 4.2 com relação ao formato dos violinos para cada programa. De fato, as amostras para cada *benchmark* seguem o mesmo tipo de distribuição (unimodal, bimodal ou trimodal) comparadas com aquelas de `ES.45`, exceto para `correlation, covariance, durbin, gesummv, reg_detect, adi, jacobi2dimper` e `seidel2d`. No entanto, mesmo para estes programas as distribuições são similares.

Essa semelhança na distribuição dos *speedups* parece indicar que independente da quantidade de otimizações disponíveis para geração das sequências (45 ou 70), cada programa segue uma mesma distribuição de *speedups* quando a geração das sequências é realizada aleatoriamente.

Comparando as distribuições de `ES.45` e `ES.70` é possível perceber que não houve grande melhoria com relação ao desempenho das sequências quando se aumentou a quantidade de otimizações de 45 para 70. Na verdade, para o espaço `ES.70` a maioria dos programas obteve uma pequena piora dos resultados e para um deles, `atax`, nenhuma sequência de *speedup* foi encontrada.

A Tabela 4.6 resume a média dos *speedups* considerando todas as sequências e apenas a melhor sequência de todos os programas dos dois espaços exploratórios. O valor máximo da sequência para todos os programas é um resultado importante, pois pode ser considerado como um limite superior de quanto é possível melhorar os programas novos que serão otimizados. Devido a estratégia de seleção compilar o código de um novo programa com alguma sequência de vinda de ES, dificilmente o desempenho deste programa não visto superará o desempenho dos programas em ES. Na Tabela 4.6 as médias do desempenho das sequências de `ES.45` e `ES.70` foram parecidas, 0.956 e 0.923, respectivamente. Considerando apenas o valor da melhor sequência de cada programa, `ES.45` obteve 1.095 de *speedup* médio e `ES.70` obteve 1.093.

Esp. Exploratório	**Todas sequências**	**Melhor Sequência**	**(%) Melhoradas**
`ES.45`	0.956	1.095	54.64%
`ES.70`	0.923	1.093	59.81%

Tabela 4.6: Valores médio e máximo de *speedups* e porcentagem de sequências melhoradas com o Algoritmo 1 para todos os programas do Polybench.

Esses resultados indicam que para a suíte Polybench, o aumento da quantidade de otimizações de 45 para 70 não afeta significativamente o desempenho das sequências geradas. Esta constatação é interessante, pois evidencia que a escolha de otimizações significantes, como aquelas pertencentes as sequências padrão do compilador, mesmo que em menor quantidade, pode trazer benefícios, como reduzir um espaço de busca de 70^{50} para 45^{40}.

A Tabela 4.6 também apresenta em sua última coluna a porcentagem de sequências melhoradas com o Algoritmo 1. Mais da metade das fases obtiveram melhor desempenho com a aplicação deste algoritmo.

4.3 Avaliação do Módulo de Seleção de Fase

Nesta seção os resultados obtidos com o Módulo de Seleção de Fase são apresentados, uma comparação com outra abordagem da literatura é fornecida, além de uma completa avaliação das estratégias de clusterização.

A Seção 4.3.1 apresenta o desempenho alcançado pelo MSF considerando os *speedups* e a quantidade de avaliações e fornece uma avaliação dos agrupamentos realizados. A Seção 4.3.2 analisa comparativamente o desempenho dos espaços exploratórios com o MSF. Por fim, a Seção 4.3.3 compara o MSF com outra estratégia de clusterização.

4.3.1 Algoritmos de Clusterização

Speedups

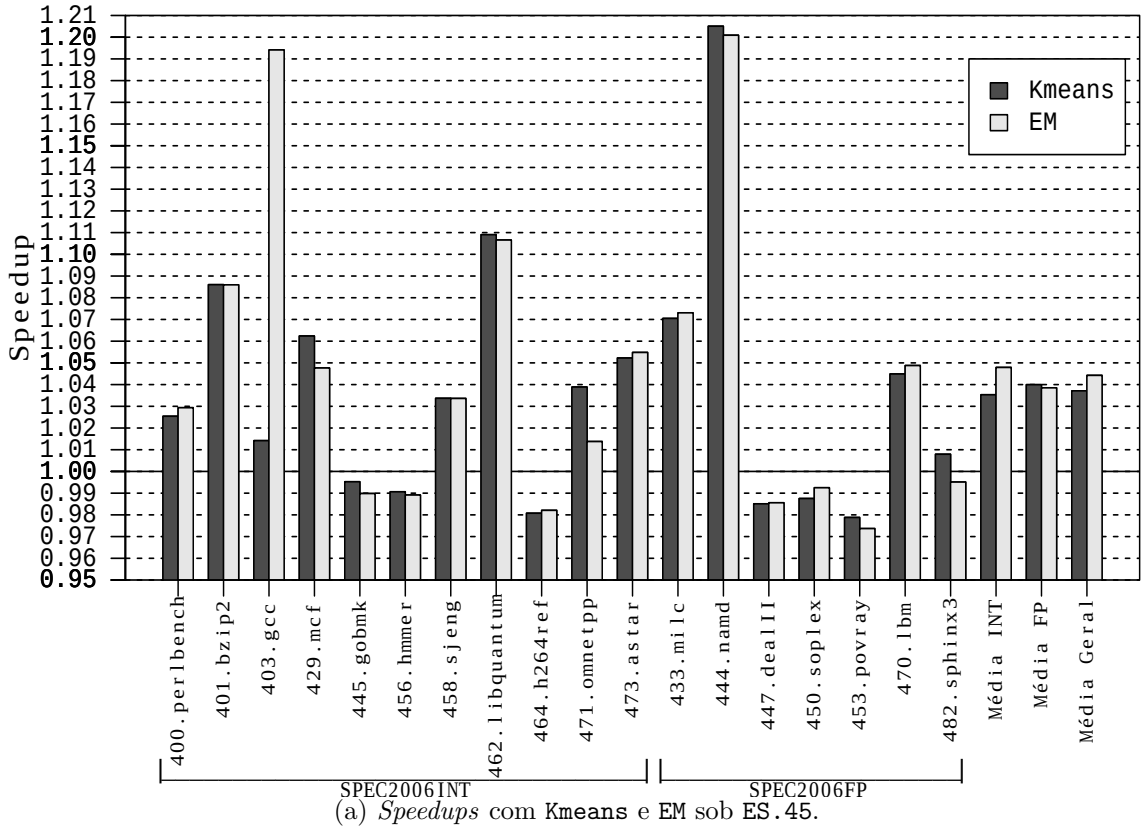
Esta seção compara os *speedups* obtidos com as estratégias `Kmeans` e `EM` empregadas no agrupamento dos programas. A Figura 4.4 apresenta os *speedups* alcançados pelos programas do SPEC2006 com o MSF executando cada um dos algoritmos de clusterização sobre os espaços exploratórios `ES.45` e `ES.70`.

A primeira observação com relação a esta figura é que, com os dois algoritmos, a maioria dos programas teve um desempenho melhor do que com o *baseline*. De fato, apenas quatro (`456.hmmer, h264ref, 447.dealII` e `450.soplex`) dos 18 programas não obtiveram melhoria com qualquer estratégia. Sobre o espaço `ES.45`, o `Kmeans` obteve um ganho médio de 03.71% e o `EM` 04.42%. Sobre o espaço `ES.70`, o `Kmeans` apresentou ganho médio de 10.76% e o `EM` de 11.86%. Nota-se também que alguns programas apresentaram grande melhoria, como o `403.gcc, 458.sjen` e `444.namd` e principalmente o `453.povray` que obteve *speedups* de 2.77 e 2.84 com o `ES.70`.

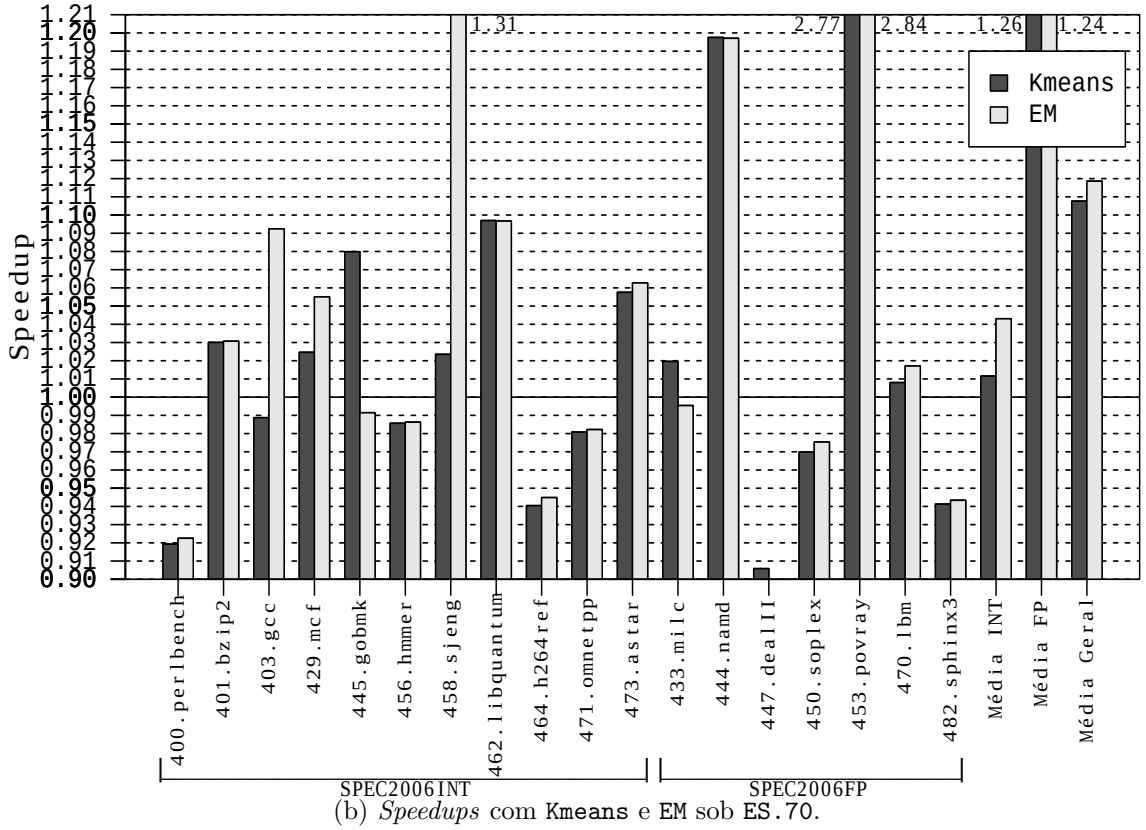
Comparando os resultados dos dois algoritmos de clusterização observa-se que não houve grande diferença de desempenho entre as duas estratégias. `Kmeans` e `EM` aumentaram o desempenho para quase os mesmos programas e obtiveram um ganho médio similar. Sobre o `ES.45` houve apenas um programa que teve uma diferença significativa de desempenho, o `403.gcc`. No espaço `ES.70` o `403.gcc` e `458.sjeng` tiveram um desempenho superior com `EM`, mas em contrapartida, `445.gobmk` e `447.dealII` foram melhores com o `Kmeans`.

A Figura 4.5 apresenta os *speedups* alcançados pelos programas da suíte cBench. De todos os 29 programas, apenas o `bitcount` e o `lame` não foram otimizados por qualquer estratégia. Sobre o espaço `ES.45`, o ganho médio obtido com o `Kmeans` foi de 190.96% e com o `EM` 198.90%. Sobre o espaço `ES.70`, o ganho médio foi de 185.11% com o `Kmeans` e 02.81% com o `EM`. É interessante notar os valores de *speedup* muito altos obtidos pelo programa `stringsearch1`: para três dos casos ele apresentou ganhos de 5418.03%, 5659.70% e 5280.16%. Estes *speedups* influenciaram o resultado médio, pois, de fato, sem considerar o `stringsearch1`, o experimento com `Kmeans` e `ES.45` obteria ganho de 04.31%, com o `EM` `ES.45` obteria ganho de 03.88%, com `Kmeans` e `ES.70` obteria ganho de 03.14% e com `EM` e `ES.70` obteria ganho de 03.08%.

Assim como ocorreu com os programas do SPEC2006, para os programas do cBench e com o espaço `ES.45`, o desempenho dos dois algoritmos de clusterização foi semelhante. Com o espaço `ES.70` o desempenho médio do `Kmeans` foi muito superior àquele com o `EM`, porém este resultado foi influenciado principalmente pelo desempenho do programa

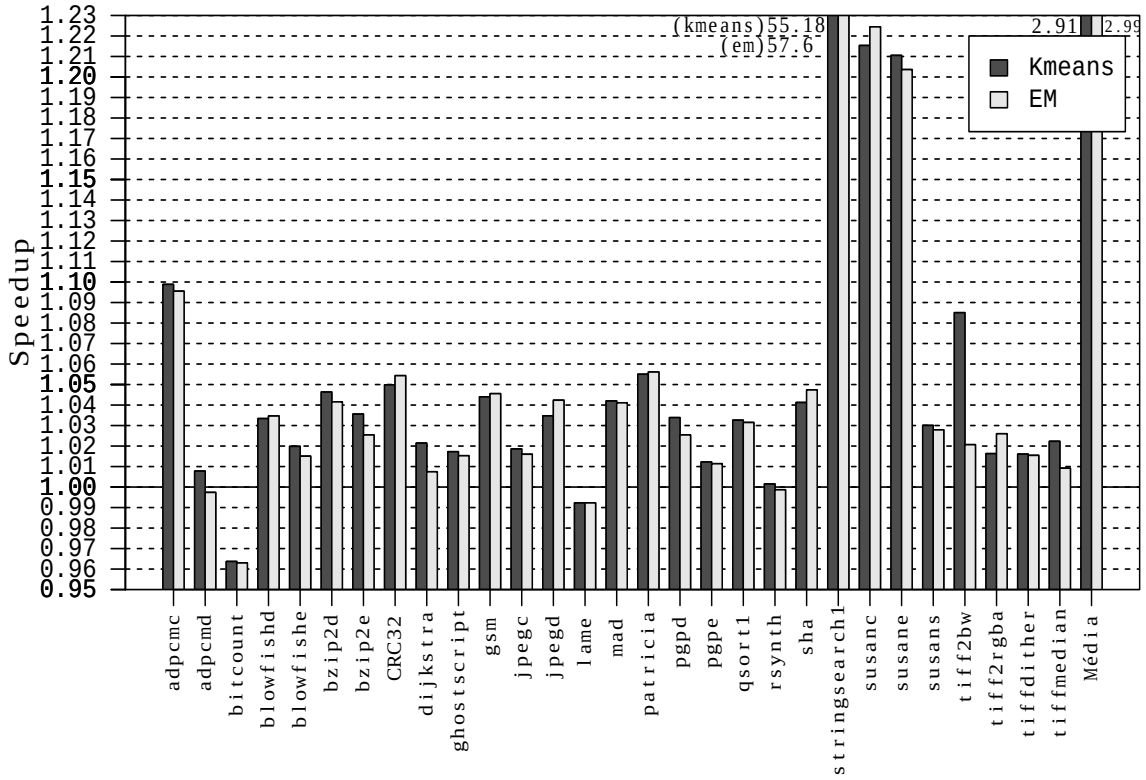


(a) *Speedups* com `Kmeans` e `EM` sob `ES.45`.

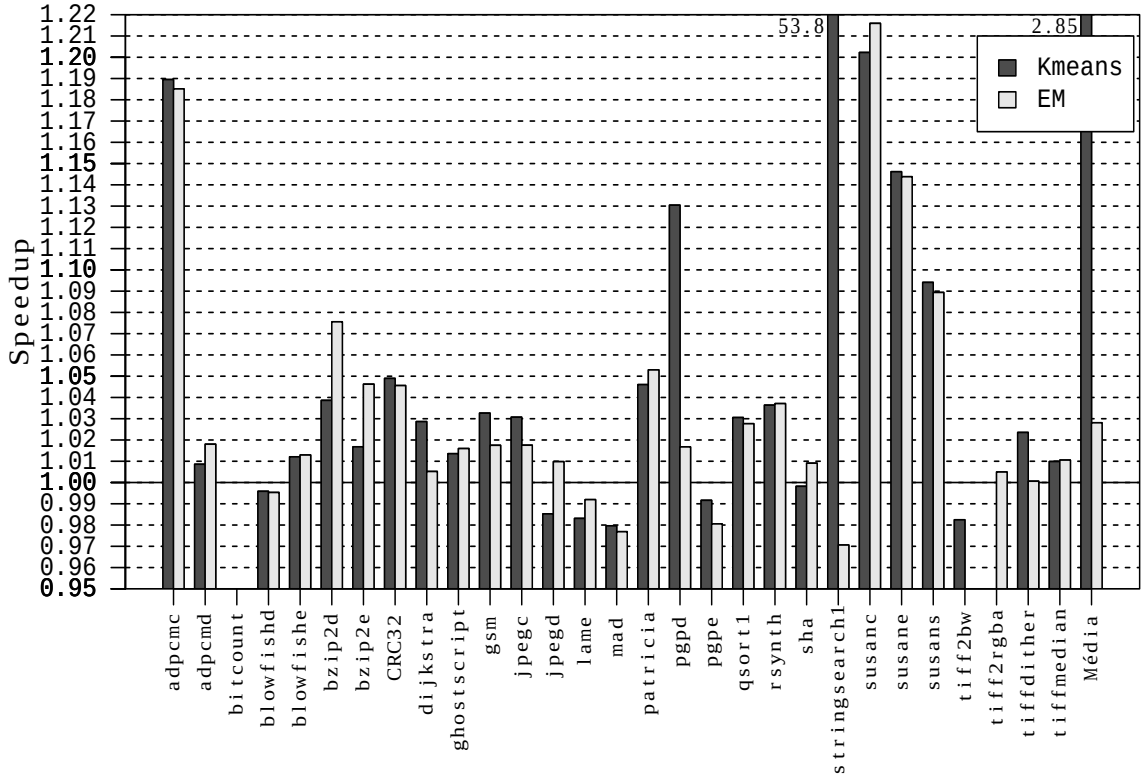


(b) *Speedups* com `Kmeans` e `EM` sob `ES.70`.

Figura 4.4: *Speedups* com `Kmeans` e `EM` sobre `ES.45` e `ES.70` para o SPEC2006.



(a) *Speedups* com `Kmeans` e `EM` sob `ES.45`.



(b) *Speedups* com `Kmeans` e `EM` sob `ES.70`.

Figura 4.5: *Speedups* com `Kmeans` e `EM` sob `ES.45` e `ES.70` para o cBench.

`stringsearch1`, o qual o `Kmeans` encontrou uma sequência que forneceu um *speedup* de 53.8 e o `EM` não. Se o `stringsearch1` for desconsiderado, o desempenho dos dois algoritmos sobre o espaço `ES.70` não apresenta diferenças significativas.

Estes resultados indicam que as duas estratégias de clusterização possibilitaram um aumento de desempenho consistente para as duas suítes de *benchmarks*. Do total de 47 programas do conjunto de teste, apenas 6 deles não apresentaram qualquer melhoria. Alguns dos programas alcançaram excelentes resultados de *speedup*, com relevância principalmente para o `453.povray` e `stringsearch1`. O primeiro chegou a apresentar um ganho de quase três vezes e o segundo um ganho de mais de 56 vezes com relação à sequência padrão do compilador.

Outro ponto importante a ressaltar é que a estratégia de clusterização, com relação ao desempenho das sequências, não influenciou o desempenho dos programas. Tanto `Kmeans` quanto `EM` apresentaram resultados de *speedup* similares.

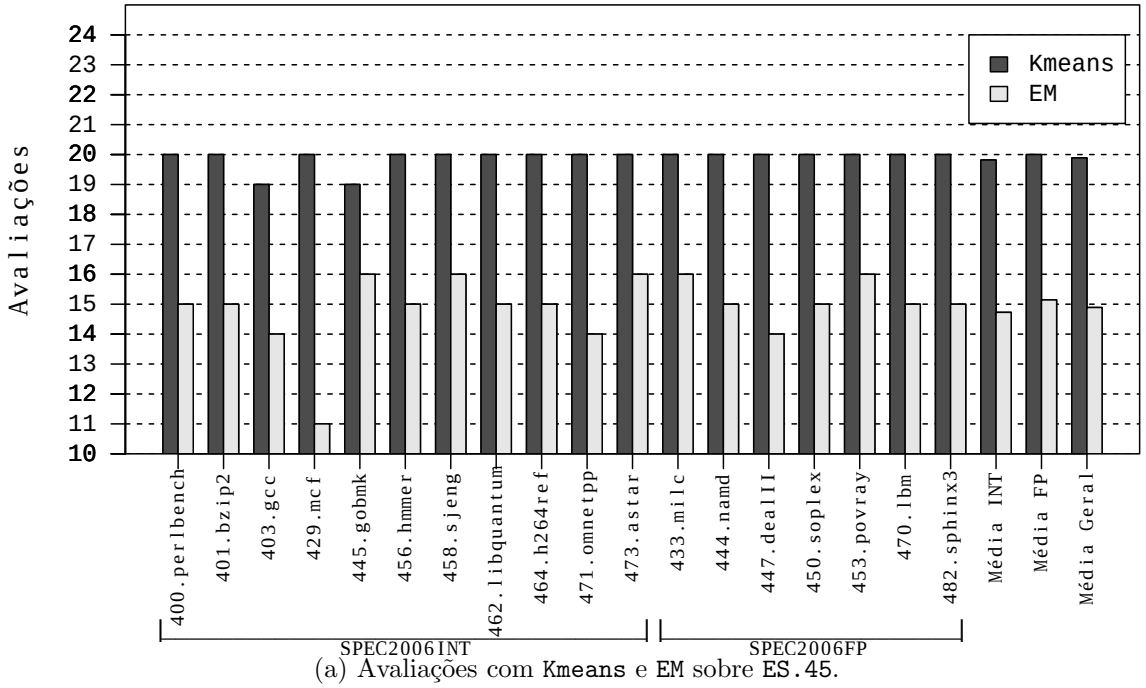
Número de avaliações

Apesar do desempenho com os dois algoritmos de clusterização terem sido similares, ao ser avaliada a quantidade de avaliações realizadas pelo MSF com cada uma destas abordagens, é possível perceber que o `EM` necessita avaliar uma quantidade menor de programas para obter este desempenho similar ou, para alguns casos, superior ao `Kmeans`.

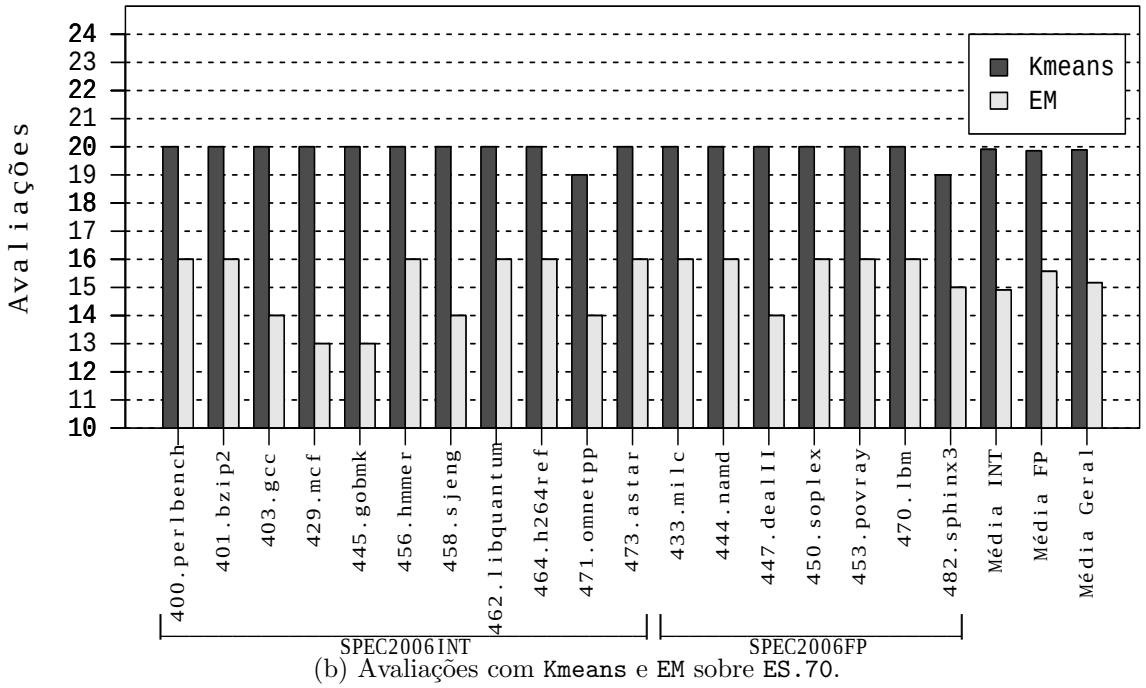
A Figura 4.6 exibe a quantidade de vezes que cada programa do SPEC2006 foi avaliado. Na média, sobre `ES.45`, o MSF realizou 19.89 avaliações com o `Kmeans` e 14.89 com o `EM`. Já sobre o `ES.70`, 19.89 avaliações foram realizadas com o `Kmeans` e 15.17 com o `EM` na média. Esses valores representam uma redução da quantidade de avaliações de 25.14% para o `ES.45` e 23.74 % para o `ES.70`, apenas mudando o algoritmo de clusterização.

A Figura 4.7 apresenta a quantidade de avaliações para cada programa do cBench. O MSF realizou uma média de 19.97 avaliações com o `Kmeans` e 15.07 com o `EM` sobre o espaço `ES.45`. Quando o espaço foi o `ES.70`, o valor médio de 19.97 avaliações foram realizadas com o `Kmeans` e 15.45 com o `EM`. O percentual de redução do número de avaliações nestes casos foi de 24.53% e 22.63% para `ES.45` e `ES.70`, respectivamente.

Por necessitar de uma quantidade menor de avaliações e fornecer um desempenho equivalente ou superior ao `Kmeans`, principalmente considerando o desempenho com o SPEC2006, o `EM` mostrou ser melhor do que o outro algoritmo. Isso também demonstra que o algoritmo de clusterização pode escolher melhor os programas do espaço exploratório. A próxima seção descreve como essa escolha ocorreu.

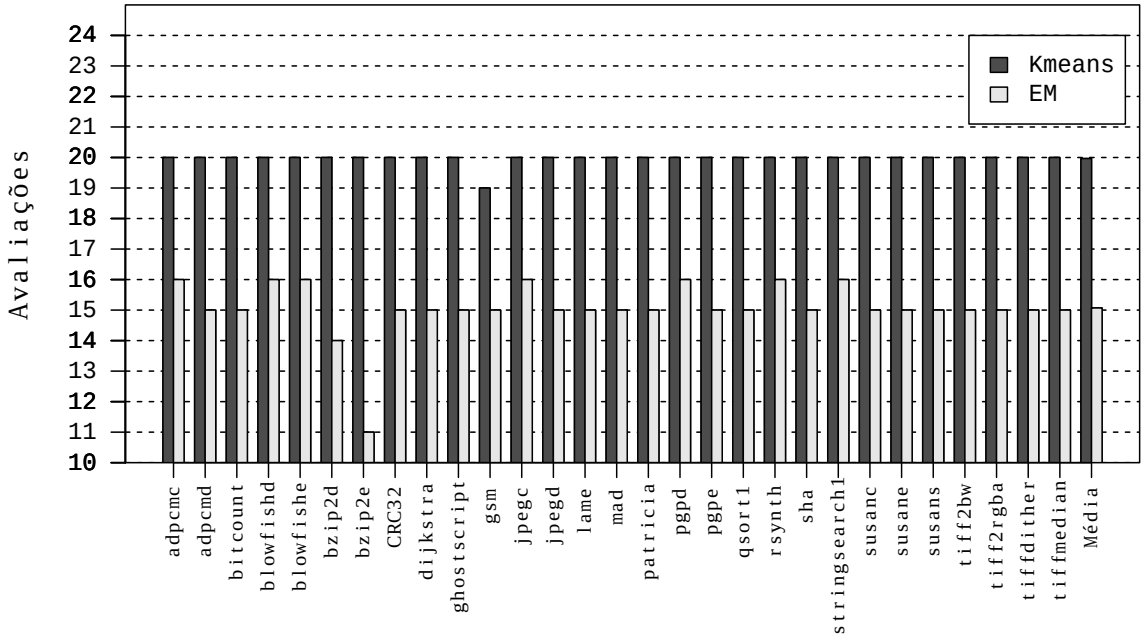


(a) Avaliações com `Kmeans` e `EM` sobre `ES.45`.

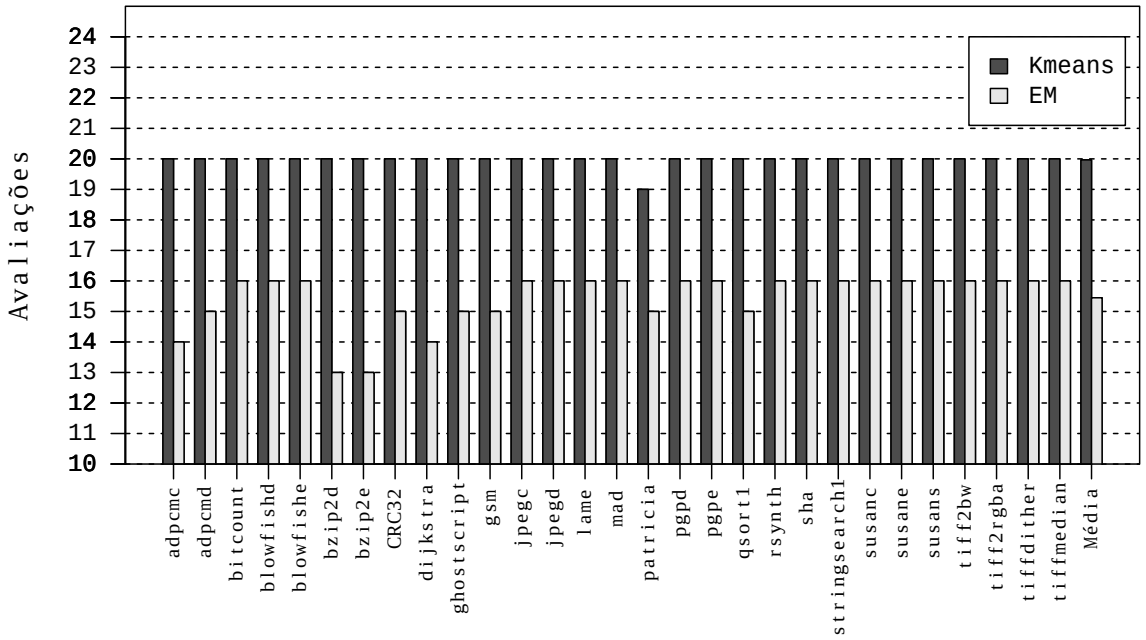


(b) Avaliações com `Kmeans` e `EM` sobre `ES.70`.

Figura 4.6: Avaliações com `Kmeans` e `EM` sobre `ES.45` e `ES.70` para o SPEC2006.



(a) Avaliações com `Kmeans` e `EM` sobre `ES.45`.



(b) Avaliações com `Kmeans` e `EM` sobre `ES.70`.

Figura 4.7: Avaliações com `Kmeans` e `EM` sobre `ES.45` e `ES.70` para o cBench.

Agrupamentos

Ao observar os gráficos da Figura 4.6 e Figura 4.7, nota-se que a quantidade de avaliações realizadas com o `Kmeans` é quase um valor constante igual a 20 e com `EM` este valor varia levemente entre 13 e 16. Esse comportamento ocorreu pois os programas do espaço exploratório que forneceram a sequência para ser usada na compilação, quase sempre foram separados nos mesmos grupos pelos dois algoritmos. Para todos os experimentos com os programas do Polybench, tanto `Kmeans` quanto `EM` separaram os programas do espaço exploratório, juntamente com o programa teste, em dois grupos que possuíam quase os mesmos programas.

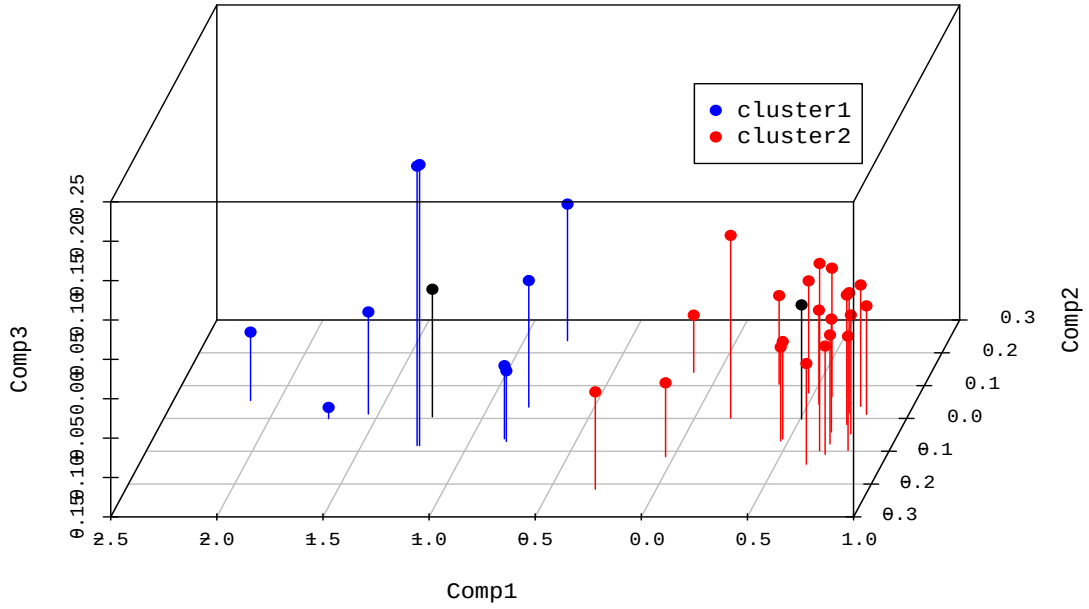
No entanto, o `EM` distribuiu os programas de maneira mais igualitária entre os dois grupos, motivo o qual possibilitou a escolha de um número menor de programas e por consequência, a redução da quantidade de avaliações.

O agrupamento dos programas em dois grupos não seria o esperado, pois se deseja que o algoritmo de clusterização seja capaz de formar pequenos grupos significativos de programas associados às melhores sequências para o programa de teste. Então, para entender o motivo porque este tipo de agrupamento ocorreu é necessário analisar como os programas estão distribuídos no espaço multidimensional. A Figura 4.8 traça os espaços tridimensionais com todos os programas do Polybench para os dois algoritmos de clusterização. O ACP reduziu a quantidade de *performance counters* para três componentes principais, por esse motivo foi possível visualizar o espaço tridimensional de programas.

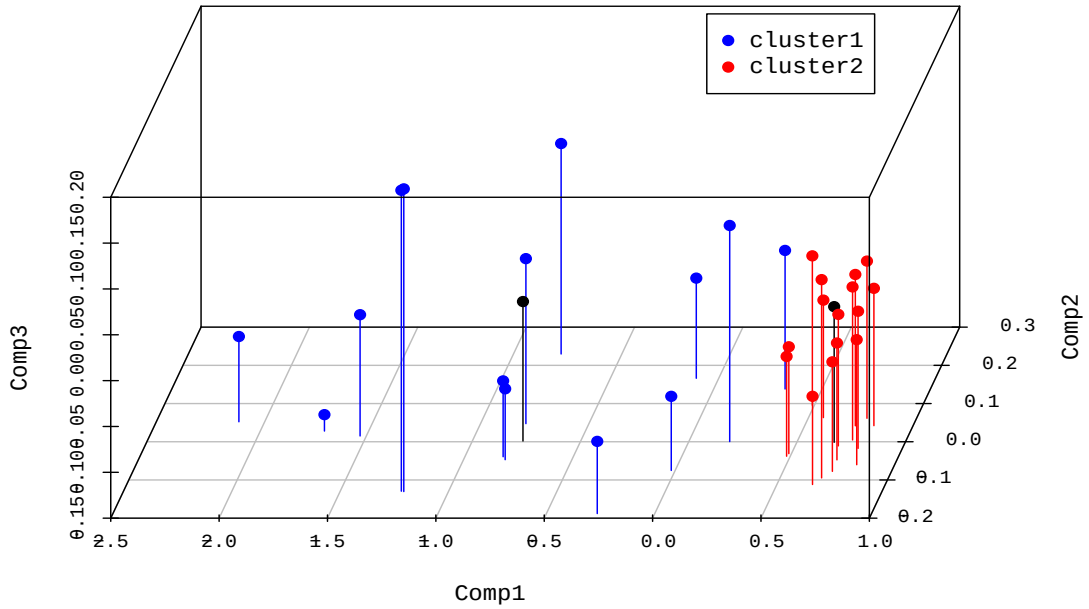
A Figura 4.8(a) retrata o agrupamento realizado pelo `Kmeans` e a Figura 4.8(b) retrata o agrupamento realizado pelo `EM`. Os dois algoritmos criaram dois *clusters* e os dois pontos pretos representam os centroides dos *clusters*. O ponto preto na nuvem de pontos azul representa o centroide do `cluster1` e o ponto preto na nuvem de pontos vermelhos representa o centroide do `cluster2`.

Ao observar a posição dos programas no espaço tridimensional da Figura 4.8 é possível perceber que existe uma alta concentração de pontos vermelhos em uma região muito pequena do espaço. Isso indica que grande parte dos programas do Polybench apresentaram características de comportamento dinâmico muito semelhantes. Essa semelhança também ocorreu com os programas de entrada, pois a maioria deles foi atribuída ao `cluster2`.

A maneira como o `Kmeans` atuou foi criar um grupo com estes programas semelhantes e outro grupo com os programas mais distantes (`cluster1`). Na Figura 4.8(b) o `EM` realizou o agrupamento de maneira semelhante, no entanto ele atribuiu alguns dos programas que não estavam muito próximos ao centroide do `cluster1` no `cluster2`. Esse agrupamento



(a) Agrupamento realizado com o `Kmeans` com os *benchmarks* do Polybench.



(b) Agrupamento realizado com o `EM` com os *benchmarks* do Polybench.

Figura 4.8: Agrupamentos com `Kmeans` e `EM` sob `ES.45` e `ES.70` com *benchmarks* do Polybench.

do EM evidenciou ser mais eficiente do que aquele com o Kmeans, pois os programas que foram atribuídos ao cluster2, na grande maioria das vezes, não forneceram sequências que aumentassem o desempenho do programa de entrada, já que os dois algoritmos apresentaram desempenho semelhante. Isso indica que retirar alguns dos programas mais afastados do grupo do programa de teste não fez com que o desempenho diminuísse, pois os programas mais próximos ao programa de entrada forneceram melhores sequências.

Uma importante observação a ser feita é que, de fato, o algoritmo de clusterização pode influenciar na escolha dos programas associados a um novo programa de entrada. Como visto, pelo fato de atribuir ao cluster1 apenas os programas mais próximos, o algoritmo EM foi capaz de escolher programas mais significativos para fornecerem sequências. No entanto, é interessante observar que a estratégia é limitada pelos programas de ES. Se todos os pontos ficassem próximos em uma única região, não haveria nada que os algoritmos de clusterização pudessem fazer para agrupar os programas. Outro ponto relevante é que a estratégia de seleção alcançou bons resultados mesmo com os programas do Polybench se mostrando muito semelhantes entre si. Na Seção 5.1, de trabalhos futuros, é sugerido a repetição dos experimentos com outros *benchmarks* para avaliar a estratégia com programas mais diversificados.

4.3.2 Espaços Exploratórios

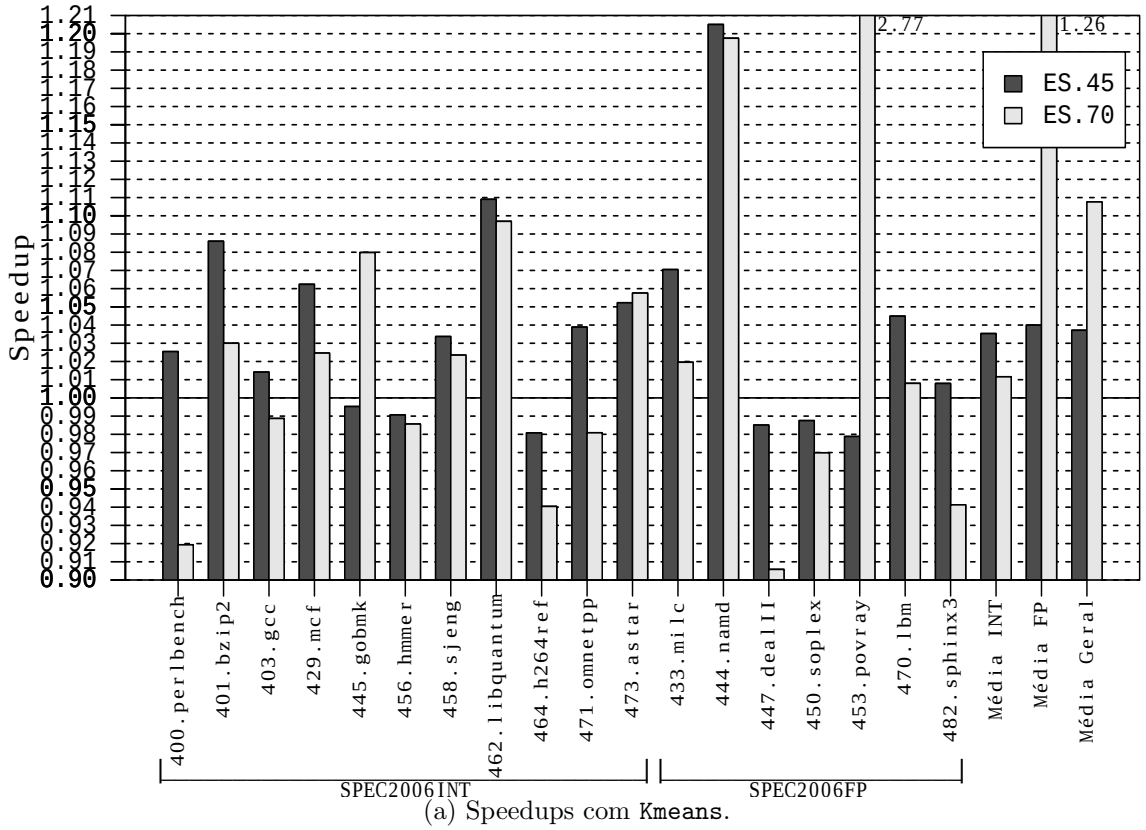
Speedups

Esta seção compara o desempenho do MSF com relação aos espaços exploratórios. Os resultados aqui são os mesmos da Seção 4.3.1, no entanto eles foram dispostos de uma maneira a comparar o desempenho de ES.45 e ES.70.

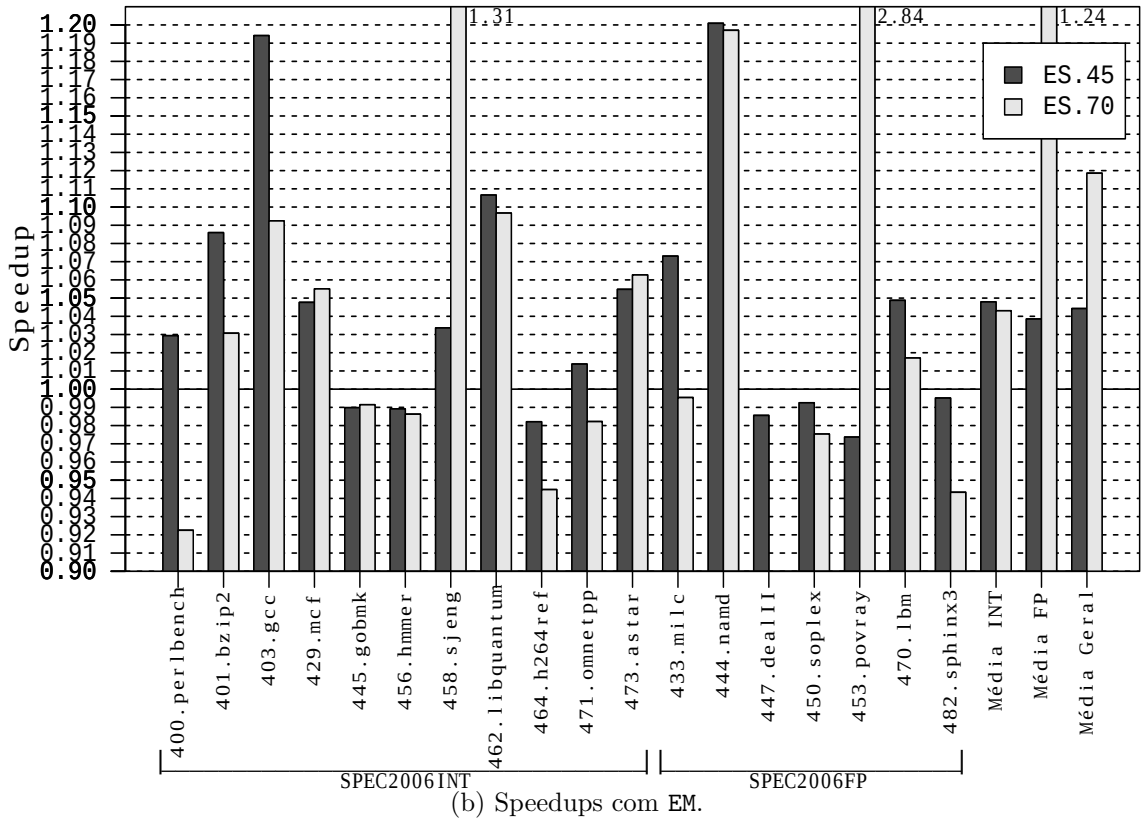
A Figura 4.9 apresenta os resultados do MSF para a coleção SPEC2006, executando com o Kmeans e EM. Para a maioria dos *benchmarks* os *speedups* com ES.45 foram superiores do que com ES.70, mesmo apesar das sequências de ES.70 serem formadas com uma quantidade maior de otimizações.

Na Figura 4.9(a) pode ser observado que 15 dos 18 programas do SPEC2006 tiveram desempenho maior com as sequências de ES.45 quando o algoritmo de clusterização foi o Kmeans, de modo que apenas o 445.gobmk e o 453.povray tiveram um desempeho superior com ES.70. Na média, o ganho alcançado com ES.45, 03.71%, foi menor do que aquele alcançado com o ES.70, 10.76%, porém este último valor foi influenciado principalmente pelo desempenho do programa 453.povray.

Com o algoritmo EM (Figura 4.9(b)), 13 dos 18 programas tiveram maior *speedup* com o ES.45. Novamente o valor de ganho médio foi muito maior com o ES.70 (11.86%) do

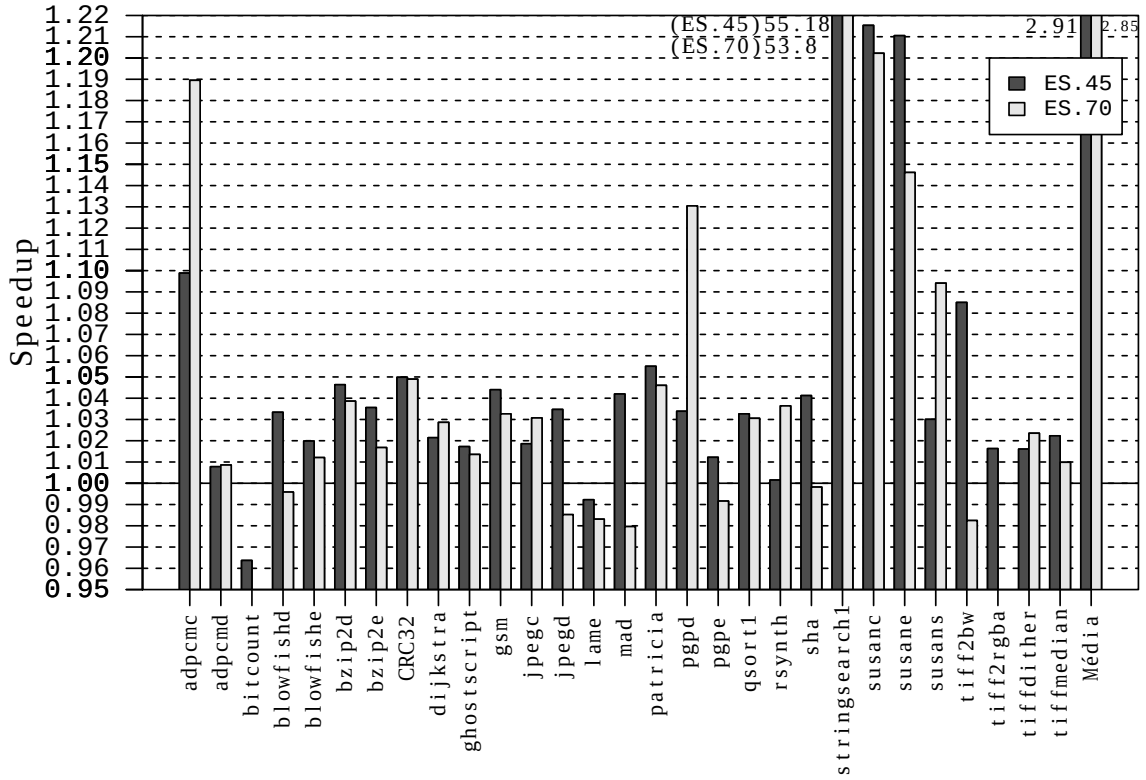


(a) Speedups com `Kmeans`.

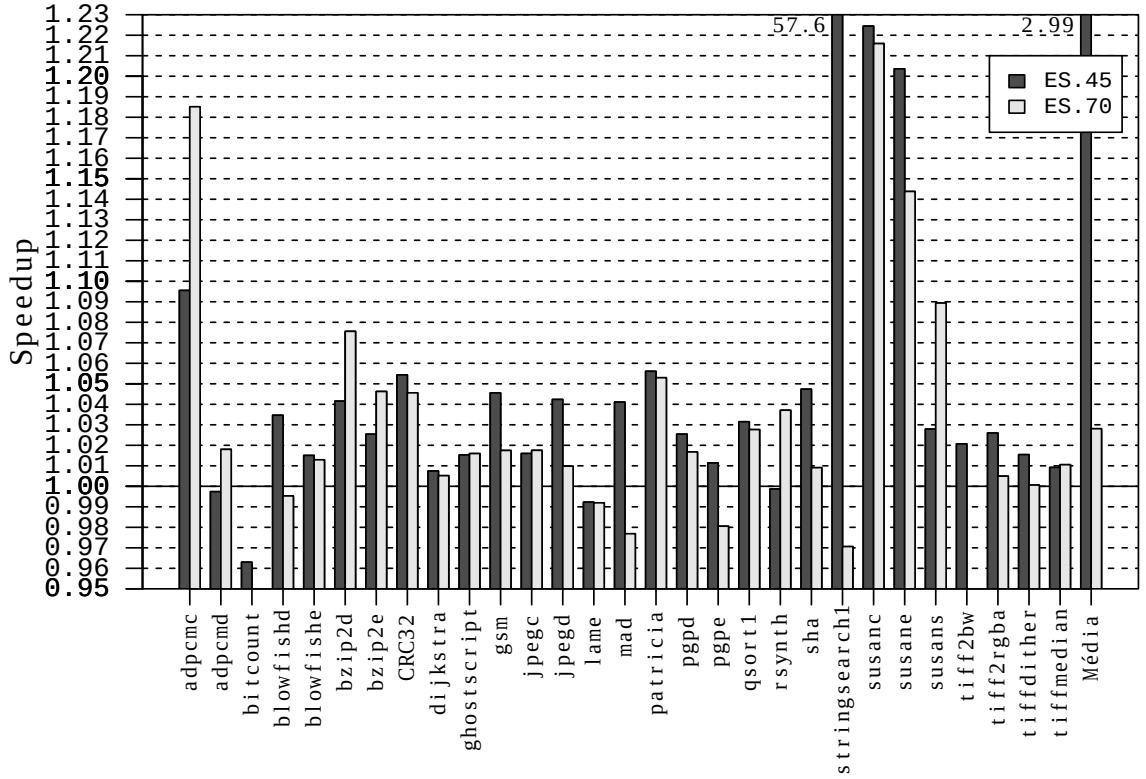


(b) Speedups com `EM`.

Figura 4.9: Comparação da estratégia de clusterização entre `ES.45` e `ES.70` com a suíte SPEC2006.



(a) Speedups com o `Kmeans`.



(b) Speedups com o `EM`.

Figura 4.10: Comparação da estratégia de clusterização entre `ES.45` e `ES.70` com a suíte cBench.

que com o ES.45 (04.42%), porém o primeiro valor foi influenciado pelo desempenho do 453.povray, como mencionado anteriormente.

Outra questão a se observar na Figura 4.9(a) é que com o espaço ES.45, seis programas não foram otimizados por nenhuma sequência e com o ES.70 esse número foi de oito programas. Na Figura 4.9(b), o resultado foi semelhante, com o ES.45 sete programas não foram otimizados e com o ES.70 nove programas não apresentaram melhoria com relação a sequência padrão.

A Figura 4.10 apresenta a comparação dos *speedups* dos espaços exploratórios com os programas do cBench. Na Figura 4.10(a), estão os *speedups* com o algoritmo Kmeans. Para 21 dos 29 programas, o maior *speedup* foi alcançado com ES.45. Com este espaço, apenas dois programas não foram otimizados (bitcount e lame) e com o espaço ES.70, oito programas tiveram *speedup* menor do que 1. Na média, os desempenhos de ES.45 e ES.70 foram semelhantes, de maneira que ES.45 obteve ganho médio de 190.96% e ES.70 de 185.11%.

Com o algoritmo EM (Figura 4.10(b)) os resultados com o ES.70 foram um pouco melhores, mas não suficientes para superar o ES.45. Dos 29 programas do cBench, 19 apresentaram maior *speedup* com ES.45 e 10 com o ES.70. Na média, o ES.45 foi muito melhor com um ganho de 198.90% contra 02.81% do ES.70. No entanto este valor foi enviesado principalmente pelo desempenho do programa stringsearch1, o qual teve um *speedup* alto de 57.6 com uma das sequências vindas de ES.45, ao passo que nenhuma sequência conseguiu otimizar este programa desta maneira com o espaço ES.70.

Estes resultados indicam que o desempenho com as sequências do espaço exploratório ES.45 foi igual ou em alguns casos superior ao desempenho das sequências do ES.70 para a maioria dos programas do cBench. Essa constatação indica que uma quantidade muito grande de otimizações na geração de sequências não necessariamente implica em um aumento de desempenho.

4.3.3 Comparação

Esta seção compara os resultados alcançados pelo MSF com a abordagem proposta por Thomson et. al (Thomson et al., 2010), a qual foi descrita na Seção 2.4.2. No entanto, primeiramente uma avaliação qualitativa dos resultados dos programas do cBench é fornecida.

Algoritmo Aleatório

Como descrito na Seção 4.3.1, a estratégia de seleção avalia aproximadamente 15 ou 20 sequências, dependendo do algoritmo, vindas do mecanismo de conhecimento prévio, o qual é criado a partir de um algoritmo aleatório. O que se deseja demonstrar nesta seção é que os resultados obtidos com a estratégia de seleção proposta são coerentes com a qualidade das sequências de ES.

Desse modo, o mesmo algoritmo para gerar as sequências de ES (Seção 3.1.1) foi empregado para gerar 500 sequências para o cBench do conjunto de teste. Dois conjuntos de 500 sequências foram criados para cada programa, de modo que no primeiro cada sequência foi gerada a partir de um conjunto de 45 otimizações e no segundo cada sequência foi gerada a partir de um conjunto de 70 otimizações.

O objetivo é fazer uma comparação da estratégia de seleção do MSF com o valor médio e o valor máximo (melhor sequência) de cada conjunto de 500 sequências. O valor máximo pode ser considerado como um limite superior para a abordagem proposta, pois dificilmente a estratégia de clusterização, que toma apenas cerca de 15 ou 20 sequências, superaria uma estratégia com 500 sequências geradas a partir do mesmo algoritmo aleatório. A Figura 4.11(a) apresenta os resultados do MSF sobre `ES.45` e a Figura 4.11(b) apresenta os resultados sobre `ES.70`. Note que na Figura 4.11(a) as 500 sequências foram geradas a partir do conjunto com 45 otimizações e na Figura 4.11(b) com o conjunto de 70 otimizações.

Para quase a totalidade dos programas na Figura 4.11, as estratégias do MSF ficaram abaixo de `melhor500`. No entanto é razoável considerar que exceto para alguns programas, como `bitcount, gsm, mad` e `pgpe`, o desempenho do MSF (`Kmeans` ou `EM`) não ficou muito distante de `melhor500`. De fato, 26, para `ES.45`, e 22, para `ES.70`, dos 29 programas obtiveram desempenho de pelo menos 90% do desempenho obtido pelo `melhor500`. Além disso, para alguns programas, como o `CRC32` e `pgpd`, a estratégia de clusterização conseguiu superar aquela `melhor500`, porém utilizando muito menos sequências. Outra observação é a de que todos os programas ficaram acima do valor médio das 500 sequências.

Este experimento ajuda a confirmar que o desempenho de uma estratégia de seleção que faz uso de uma estrutura como o espaço exploratório é extremamente dependente da qualidade das sequências presentes no mecanismo de conhecimento prévio. Além disso, a Figura 4.11 possibilita evidenciar que a estratégia de clusterização proposta é capaz de encontrar boas sequências e que os *speedups* delas são coerentes com aqueles do espaço exploratório. É deixado como um trabalho futuro a realização deste experimento com

os programas do SPEC2006 para verificar se este comportamento se apresenta com esta suíte de *benchmarks*.

Algoritmo de Thomson

Nesta seção os resultados do MSF são comparados com a estratégia de clusterização proposta por Thomson et al. (Thomson et al., 2010), a qual foi implementada como descrito na Seção 2.4.2. A única diferença entre esta implementação e a original foi a utilização dos mesmos espaços exploratórios do MSF como mecanismo de conhecimento prévio do trabalho de Thomson et al. Dessa forma as duas estratégias podem selecionar sequências da mesma base de conhecimento.

A Figura 4.12 e Figura 4.13 apresentam os *speedups* do MSF com o `Kmeans` e `EM` comparados com a estratégia de Thomson et. al (`thomson`). Observando a Figura 4.12 nota-se que o MSF obteve melhor desempenho para quase todos os *benchmarks*, com exceção de `462.libquantum` e `482.sphinx3` na Figura 4.12(a), os quais o MSF apresentou desempenho equivalente ao trabalho dos outros autores. Na Figura 4.13, apenas os programas `adpcmd, qsort1, pgpe` e `susans` não obtiveram maior *speedup* com o MOF.

A estratégia de Thomson et al. conseguiu otimizar no máximo apenas cinco programas do SPEC2006 e doze do cBench. Sua vantagem é o fato de necessitar de apenas uma avaliação para o programa, no entanto, ela não é adequada para uma otimização agressiva da aplicação.

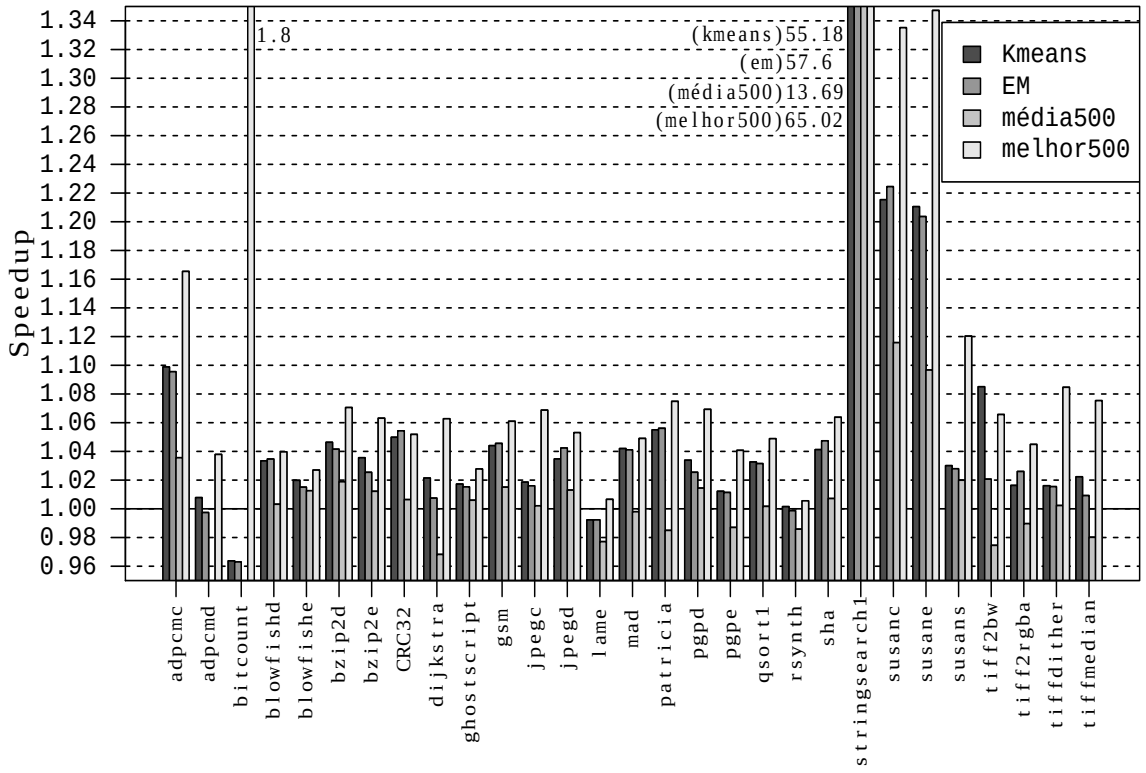
4.4 Avaliação do Módulo de Ordenação de Fase e FSOF

Esta seção avalia o desempenho do Módulo de Ordenação de Fase. Como o MOF não opera isoladamente, ela também apresenta os resultados alcançados pelo FSOF utilizando o MSF e MOF conjuntamente. Uma análise do algoritmo de transformação e do algoritmo de colônia de formigas com relação aos pares de otimizações das sequências também é fornecida.

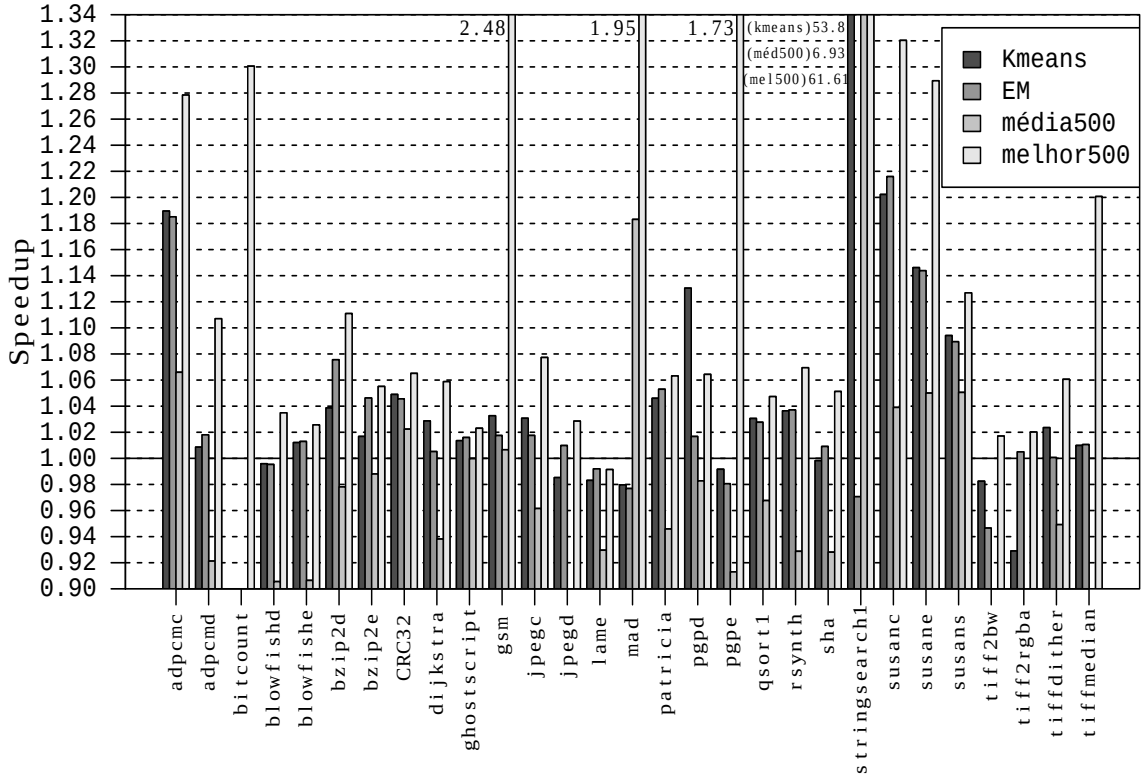
A Seção 4.4.1 apresenta o desempenho do MOF e FSOF. A Seção 4.4.2 analisa os algoritmos de transformação e de colônia de formigas para o POF. Por fim, na Seção 4.4.3 os resultados do FSOF são comparados com um algoritmo de compilação iterativa.

4.4.1 Speedups e Número de Avaliações

A Figura 4.14 e a Figura 4.15 exibem os *speedups* alcançados pelo MOF tendo como sequência de entrada a melhor sequência fornecida pelo MSF para cada experimento

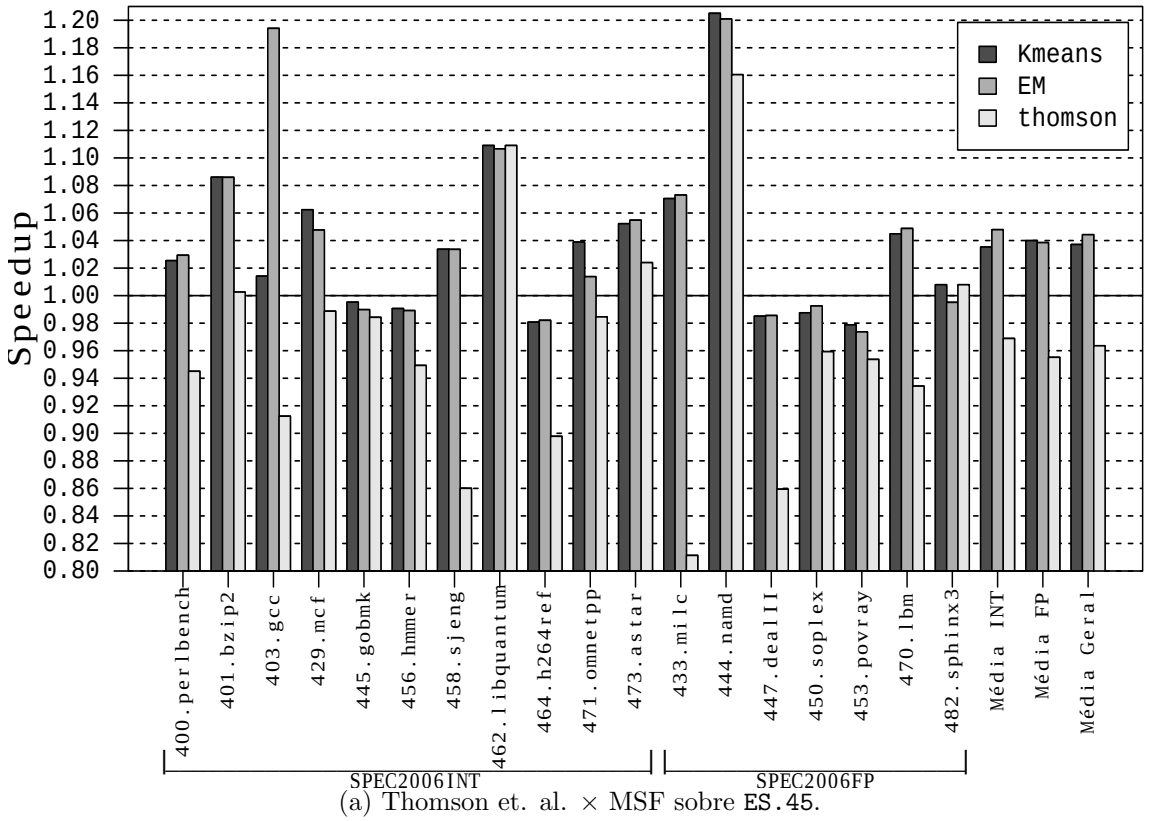


(a) Speedups sobre `ES.45`.

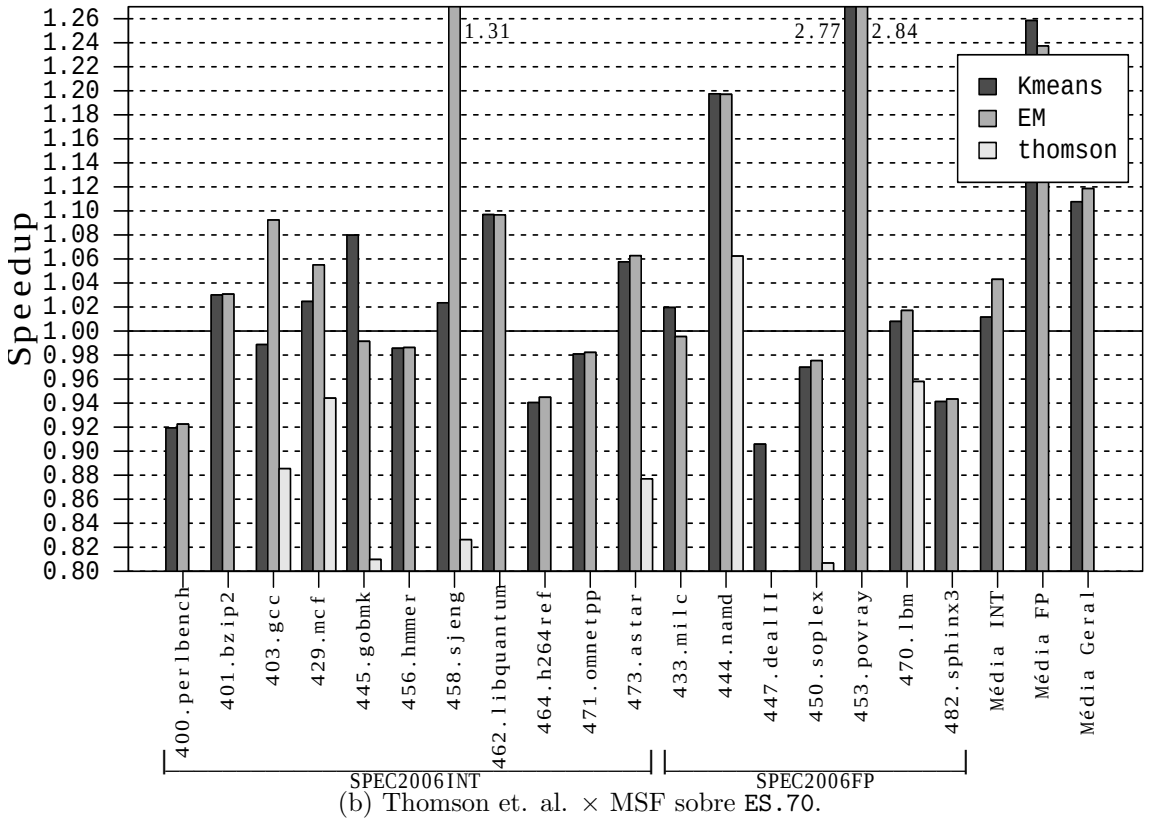


(b) Speedups sobre `ES.70`.

Figura 4.11: Comparação da estratégia de clusterização entre `ES.45` e `ES.70` com a suíte cBench.

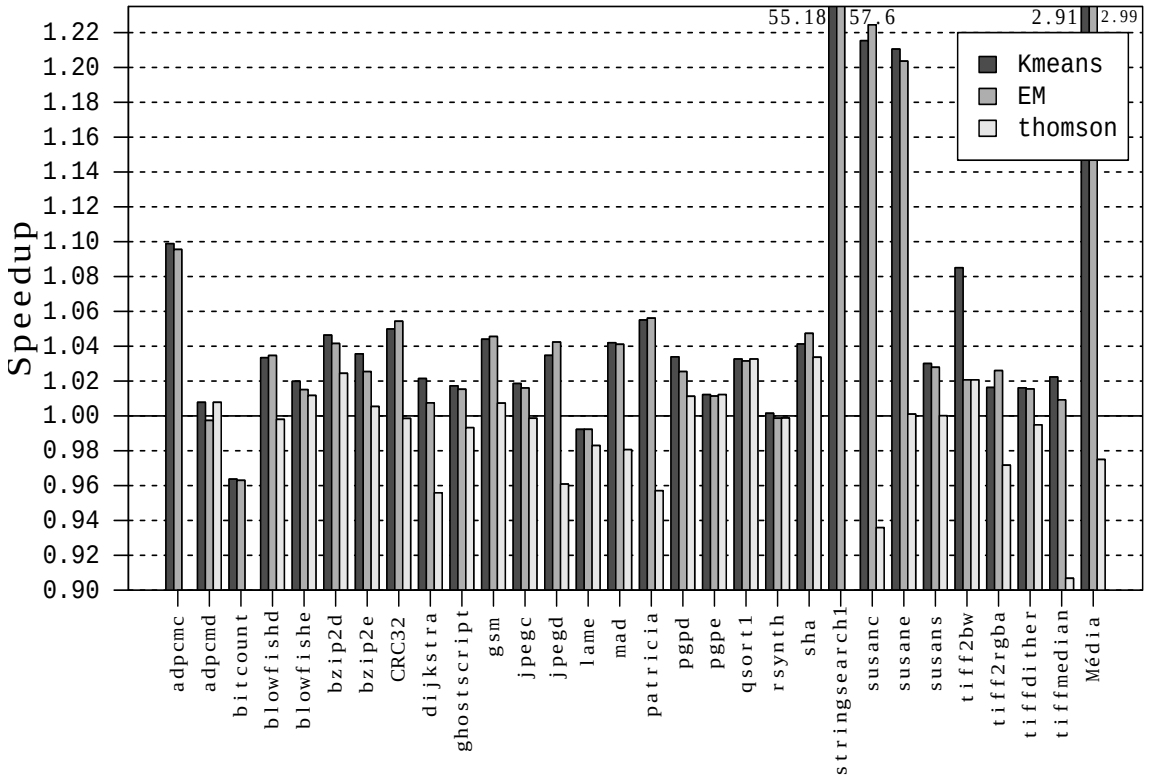


(a) Thomson et. al. × MSF sobre `ES.45`.

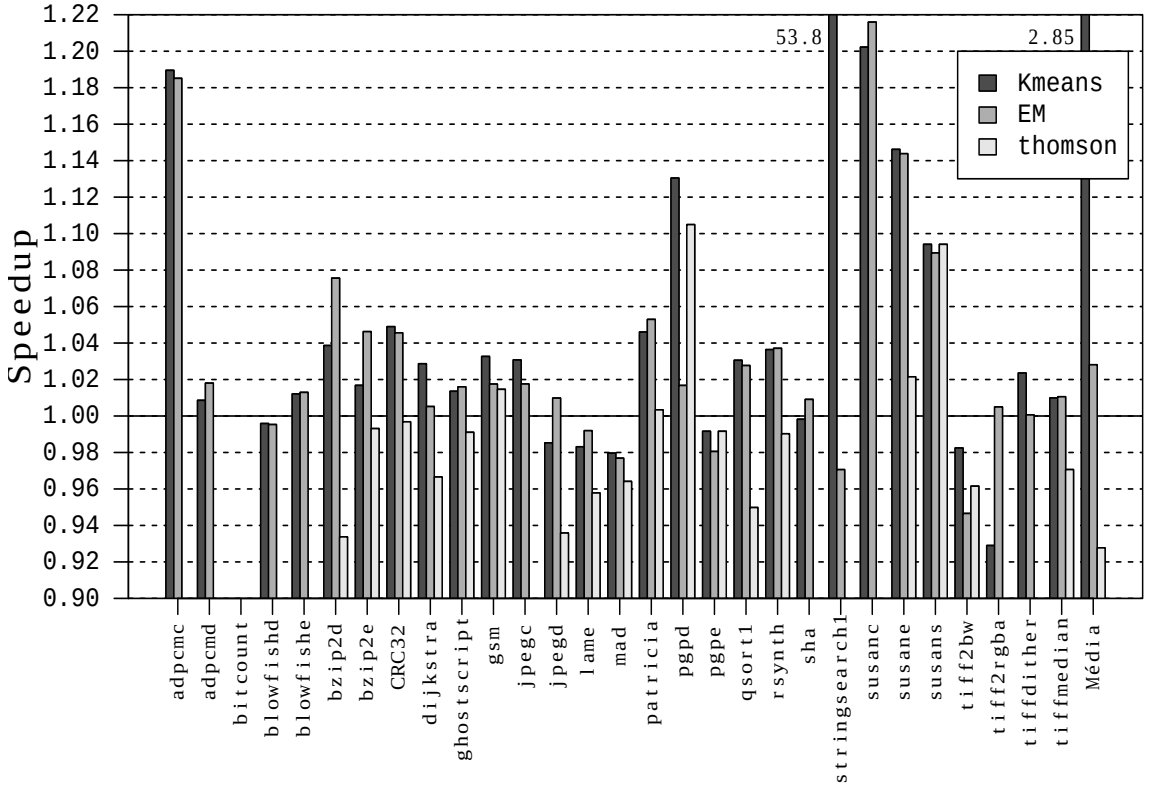


(b) Thomson et. al. × MSF sobre `ES.70`.

Figura 4.12: Comparação da estratégia de Thomson et. al × o MSF, para os *benchmarks* do SPEC2006, sobre os espaços `ES.45` e `ES.70`.



(a) Thomson et. al. × MSF sobre `ES.45`.



(b) Thomson et. al. × MSF sobre `ES.70`.

Figura 4.13: Comparação da estratégia de Thomson et. al × o MSF, para os *benchmarks* do cBench, sobre os espaços `ES.45` e `ES.70`.

realizado. Essas sequências vieram do MSF executando o `Kmeans` (Figura 4.14(a) e Figura 4.14(b)) e o `EM` (Figura 4.15(a) Figura 4.15(b)) sobre os espaços exploratórios com programas do Polybench.

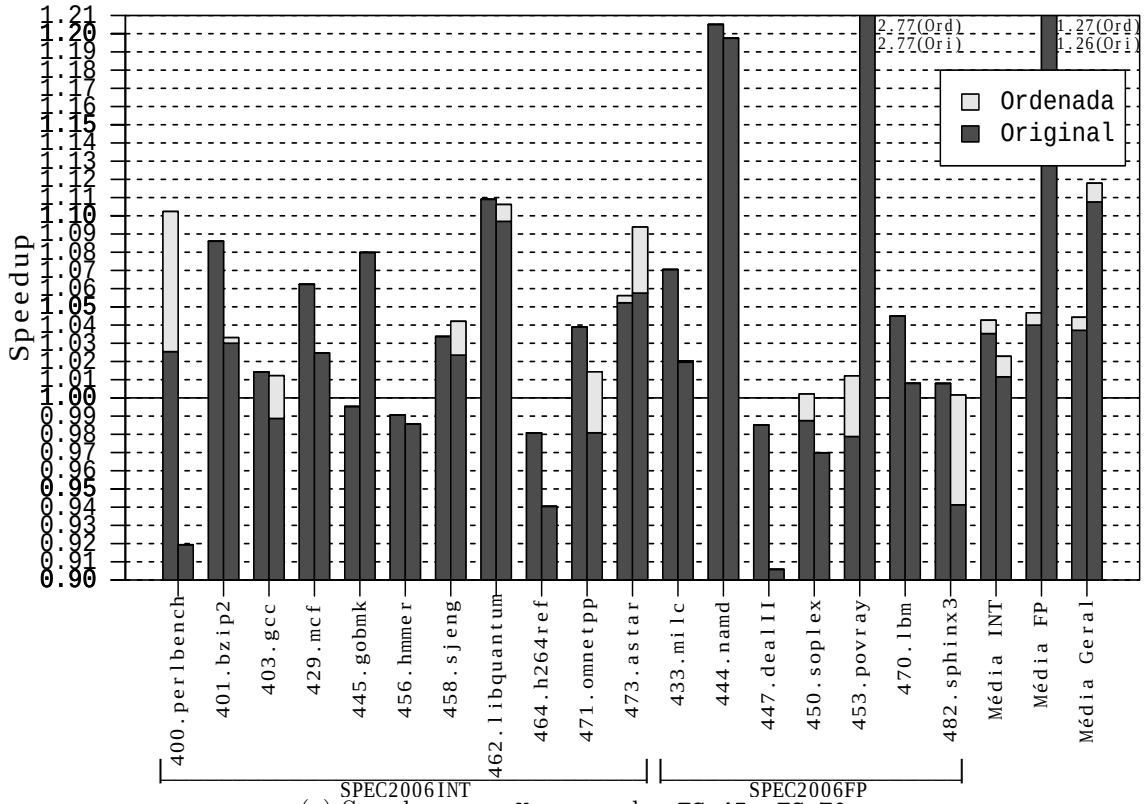
Em cada figura a primeira barra de cada programa sempre corresponde à `ES.45` e a segunda à `ES.70`. A barra referenciada por `Original` corresponde ao *speedup* da sequência de entrada vinda do MSF e a barra rotulada como `Ordenada` se refere ao *speedup* da melhor sequência gerada pelo MOF a partir da sequência `Original`. Os resultados apenas da barra `Ordenada` retratam o ganho da melhor sequência gerada pelo MOF com relação à sequência vinda do MSF, considerando o valor da parte rotulada como `Original` e também da parte rotulada como `Ordenada` obtém-se o desempenho total alcançado pelo FSOF.

Desempenho MOF

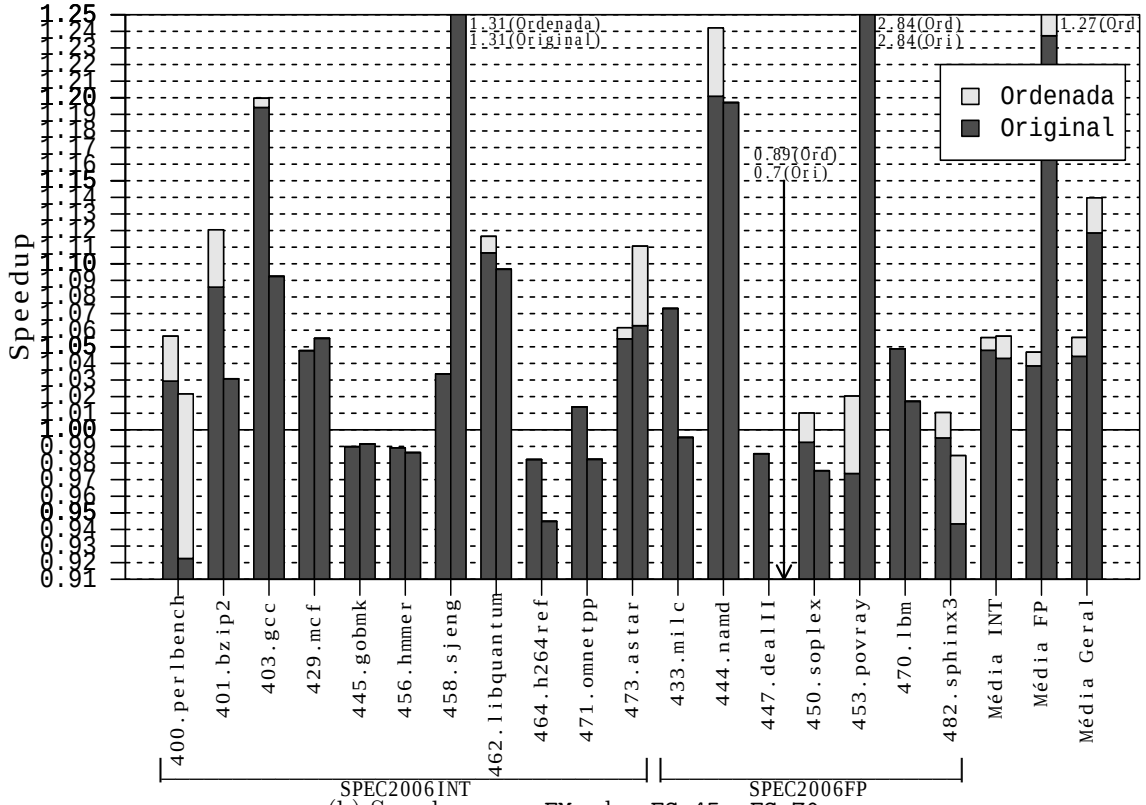
Na Figura 4.14(a), os ganhos médios com o `Kmeans` foram de 0.72% e 01.03% com `ES.45` e `ES.70`, respectivamente. Com o `EM`, Figura 4.14(b), os ganhos médios foram de 01.14% com `ES.45` e 02.11% com `ES.70`. Considerando a Figura 4.14, os programas `400.perlbench, 401.bzip2, 471.omnetpp, 473.astar, 444.namd, 453.povray` e `482.sphinx3` obtiveram um ganho de pelo menos 03% para algum dos espaços exploratórios e os demais obtiveram um ganho marginal ou não apresentaram melhoria. É interessante notar que o programa `450.soplex`, o qual o MSF não foi capaz de fornecer alguma sequência que aumentasse o seu desempenho com relação ao *baseline*, foi melhorado com os dois algoritmos de clusterização pelo MOF.

A Figura 4.15(a) exibe os *speedups* com o `Kmeans`, de modo que os ganhos médios foram de 02.54% com `ES.45` e 01.21% com `ES.70`. Na Figura 4.15(b), o ganho médio com o `EM` foi de 02.38% para `ES.45` e 0.81% para `ES.70`. Vinte e seis programas dos 29 do cBench obtiveram alguma melhoria, de modo que alguns deles alcançaram um aumento de desempenho significativo (`adpcmd, bitcount, bzip2d, stringsearch1, susane, tiff2bw, tiff2rgba` e `tiffdither`) e os demais aumentaram o *speedup* marginalmente. É importante notar que o programa `bitcount`, o qual não havia melhorado com o MSF obteve um aumento de desempenho de aproximadamente 41.41%.

Estes resultados indicam que a estratégia de ordenação aumentou o desempenho da maioria dos programas pelo menos marginalmente. Dentro de um ambiente como o FSOF, ela poderia ser usada principalmente em cenários em que há tempo disponível para execução do programa mais vezes a fim de tentar melhorar ainda mais o desempenho

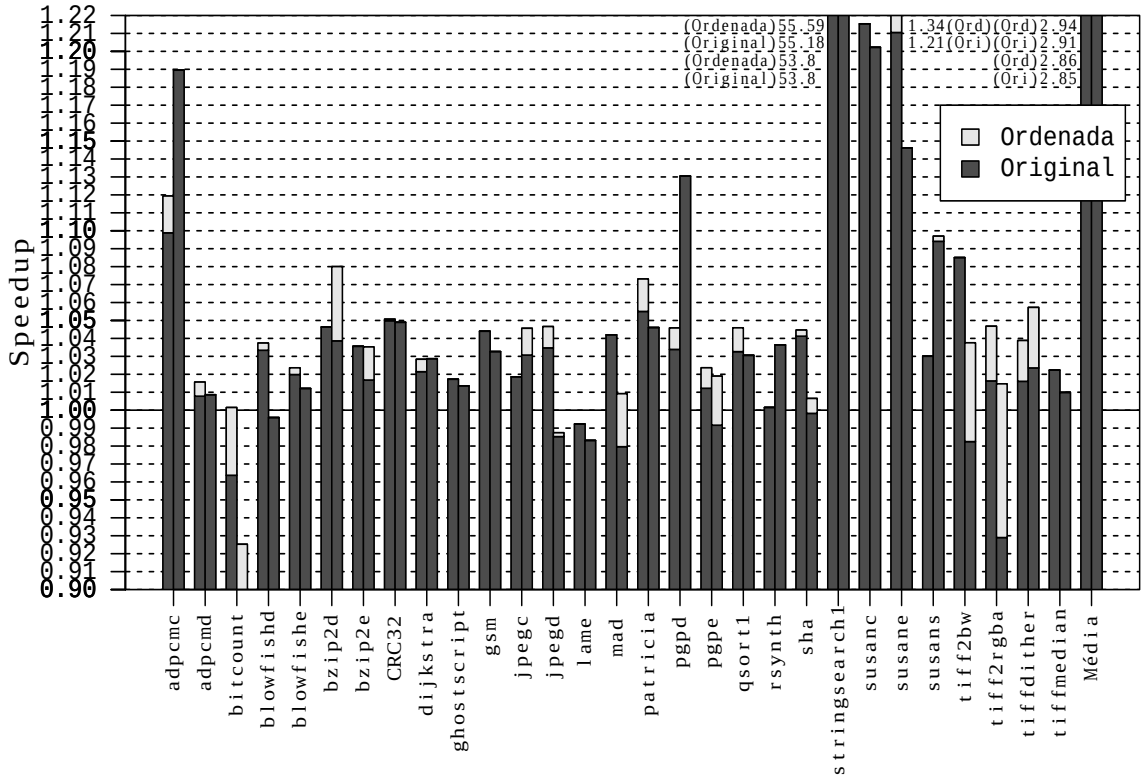


(a) Speedups com `Kmeans` sobre `ES.45` e `ES.70`.

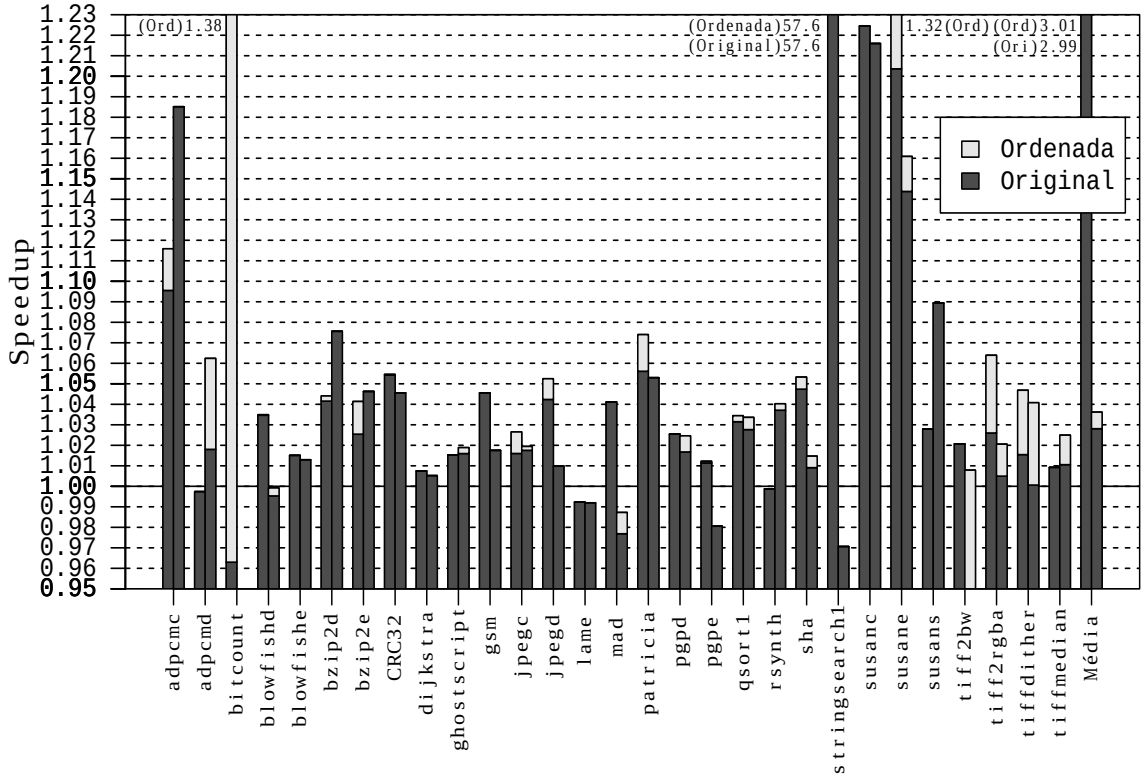


(b) Speedups com `EM` sobre `ES.45` e `ES.70`.

Figura 4.14: Speedups para os *benchmarks* do SPEC2006 variando os algoritmos de clusterização e espaços exploratórios.



(a) Speedups com `Kmeans` sobre `ES.45` e `ES.70`.



(b) Speedups com `EM` sobre `ES.45` e `ES.70`.

Figura 4.15: Speedups para os *benchmarks* do cBench variando os algoritmos de clusterização e espaços exploratórios.

obtido pelo MSF. No caso em que o tempo para gerar a versão final do programa é um fator crítico, os ganhos obtidos com o MOF não são suficientes para justificar seu uso.

Desempenho FSOF

Considerando o desempenho geral obtido pelo FSOF, para a maioria dos *benchmarks* do conjunto de teste o *framework* descobriu alguma sequência que superasse a estratégia padrão do compilador. De fato, do total de 47 programas para apenas quatro (`456.hmmer`, `464.h264ref`, `447.dealII` e `lame`) o FSOS não encontrou alguma sequência que superasse o *baseline*.

A Tabela 4.7 apresenta um resumo dos resultados para todas as estratégias. Para o SPEC2006 a melhor estratégia foi com o MSF realizando a clusterização com o `EM`, pois obteve os maiores *speedups* com uma quantidade menor de avaliações. Para o cBench, os desempenhos de *speedups* com `ES.45` foram semelhantes e o desempenho com o `ES.70.kmeans` foi superior do que com `ES.70.em`. No entanto, o `EM` sempre possibilitou menos avaliações.

É importante salientar que os ganhos de aproximadamente 200% com o cBench foram devido ao alto desempenho obtido pelo programa `stringsearch1`, como destacado ao longo de todo o texto.

Outro ponto importante é que o ambiente de compilação adaptativa implementado no FSOF pode proporcionar sequências que otimizam os programas de maneira extremamente agressiva como foram os casos dos programas `453.povray` e `stringsearch1`.

4.4.2 Pares de Otimizações

Esta seção apresenta uma análise da estratégia de ordenação com relação aos pares de otimizações das sequências geradas pelo MOF. Os dados foram extraídos de três experimentos realizados com os programas do SPEC2006, pois essa é a suíte de *benchmarks* mais representativa das duas utilizadas. A configuração dos parâmetros foi a padrão descrita na Seção 4.1.2. No entanto, para cada um dos três experimentos o parâmetro β do algoritmo de colônia de formigas foi configurado para um valor diferente: $\beta = \{1, 5, 15\}$.

O parâmetro β corresponde à importância da visibilidade de uma formiga durante a construção de uma solução. A visibilidade é uma função inversamente proporcional à distância do nó em que a formiga está posicionada com relação ao próximo nó que ela vai escolher. Quanto maior o valor da visibilidade, mais probabilidade a formiga possui de escolher um próximo nó de menor custo. Ao variar β o objetivo é inferir se o algoritmo de colônia de formigas foi adequado durante a resolução do POF. Além disso, é possível

Origem da Sequência	Ganho	GEP	GEN	GEPN	NMA	NPS
SPEC2006						
`ES.45.kmeans`	04.43%	04.62%	06.04%	06.41%	19.89 + 10	14 de 18
`ES.70.kmeans`	11.79%	02.09%	18.47%	03.41%	19.89 + 10	13 de 18
`ES.45.em`	05.56%	05.77%	07.53%	07.95%	14.89 + 10	14 de 18
`ES.70.em`	13.97%	03.99%	27.66%	06.67%	15.17 + 10	10 de 18
cBench						
`ES.45.kmeans`	193.50%	05.46%	200.43%	05.69%	19.97 + 10	28 de 29
`ES.70.kmeans`	186.31%	04.39%	216.56%	04.62%	19.97 + 10	25 de 29
`ES.45.em`	201.28%	06.34%	224.55%	07.14%	15.07 + 10	27 de 29
`ES.70.em`	03.62%	03.85%	05.33%	03.94%	15.45 + 10	23 de 29

Tabela 4.7: Desempenho do FSOF para todas as estratégias. **Ganho**: ganho médio de *speedup* para todos os programas. **GEP**: ganho médio excluindo os programas com maiores picos de *speedup* (`453.povray` para o SPEC2006 e `stringsearch1` para o cBench). **GEN**: ganho médio excluindo os programas que não alcançaram *speedup*. **GENP**: ganho médio excluindo programas pico e sem *speedup*. **NMA**: número médio de avaliações. **NPS**: número de programas que alcançaram *speedup*.

verificar se o algoritmo de transformação do MOF foi capaz de extrair algum conhecimento do espaço exploratório.

A Figura 4.16 retrata o custo dos pares escolhidos pelo MOF para construção das sequências geradas. Nesta figura, cada coluna representa a quantidade total de sequências geradas pelo MOF que continham pares de otimizações (i, j), em que a otimização que dá nome a coluna corresponde a i e j é qualquer outra otimização que não i ($j \in O - \{i\}$) que ocorreu na sequência. Por exemplo, a primeira coluna `scalarrepl` corresponde a quantidade de sequências geradas que continham em algum ponto o par (`scalarrepl`, j), para qualquer otimização j. Cada divisão das colunas corresponde a quantidade de sequências que continham a otimização i, de modo que o par que ela formava com alguma outra otimização j teve custo atribuído pelo algoritmo de transformação situado entre os intervalos $[0.00, 0.25)$, $[0.25, 0.50)$, $[0.50, 0.75)$ e $[0.75, 1.00]$.

O principal objetivo do MOF durante a ordenação da sequência consiste em escolher de uma maneira aleatória tendenciosa os pares que possuem o menor custo e que a soma geral dos custos seja minimizada. Desse modo, se espera que os pares das sequências geradas pelo módulo de seleção possuam um custo baixo. Observando a Figura 4.16 é possível notar que este objetivo é alcançado, pois para a maioria dos pares formados com as otimizações i, que representam cada coluna, o custo está entre os intervalos $[0.00, 0.25)$

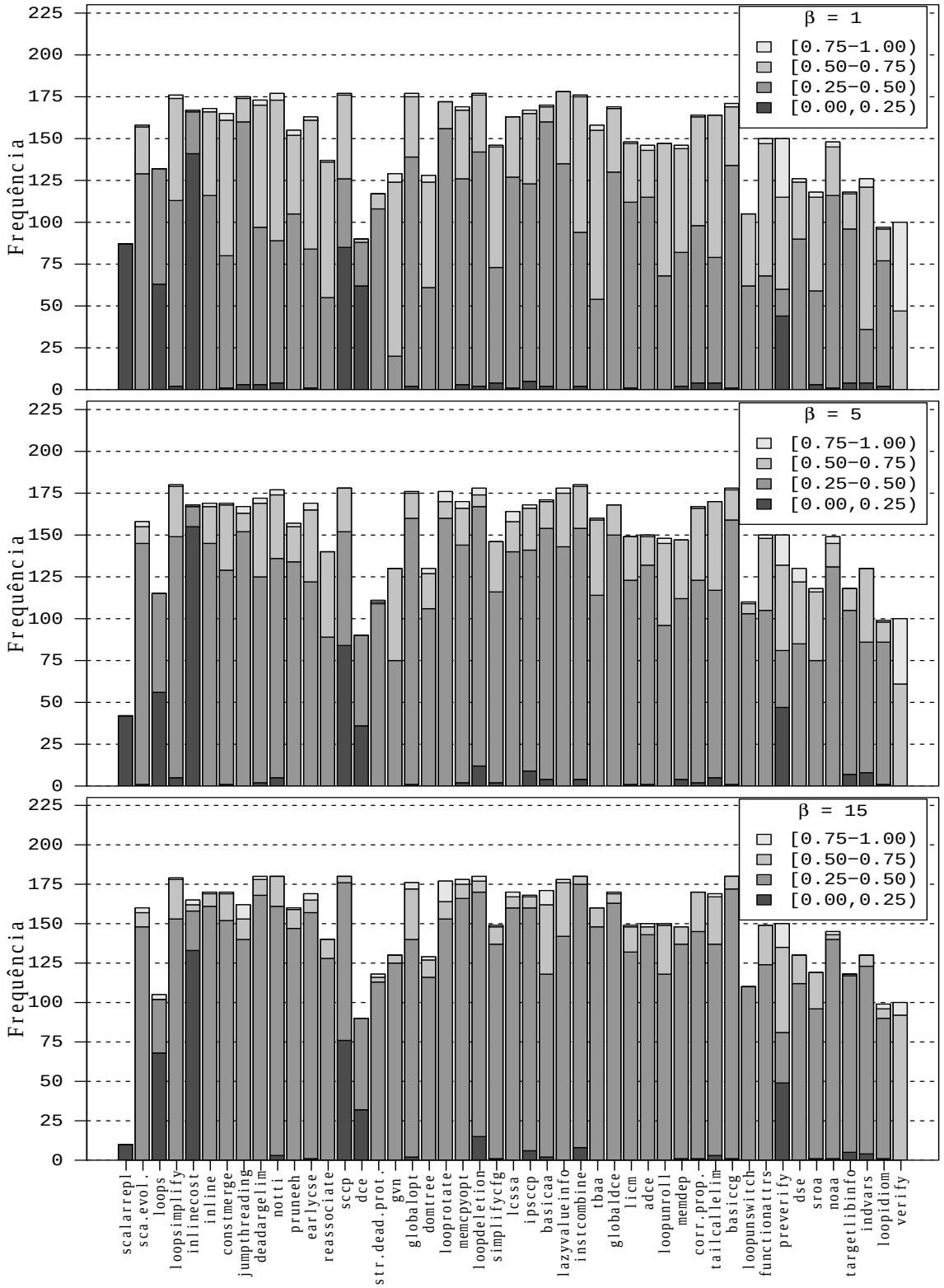


Figura 4.16: Frequência das sequências com pares (i, j) para todas as otimizações, separados por intervalos de custo e para $\beta = \{1, 5, 15\}$.

e $[0.25, 0.50)$. Isso significa que apesar da aleatoriedade, o algoritmo tende a escolher os pares que possuem menor custo.

É interessante notar que as otimizações que formaram uma maior quantidade de pares com custo entre $[0.00, 0.50]$ foram `scalarrepl, loops`, `inlinecost`, `sccp`, `dce` e `preverify`. Estas otimizações são aquelas envolvidas no algoritmo de melhoria das sequências do espaço exploratório (Algoritmo 1). Como boa parte das sequências de ES foram melhoradas por esse algoritmo, este conhecimento foi extraído pelo algoritmo de transformação e depois pelo *Ant System*, situação que foi refletida nas sequências geradas. Se for observado como exemplo a coluna `inlinecost`, a grande maioria dos pares dentro do intervalo $[0.00, 0.25]$ é (`inlinecost, inline`), o qual é descrito no Algoritmo 1 como um par potencial para melhoria da sequência. Todas as outras otimizações citadas também apresentam essa tendência do Algoritmo 1. Este comportamento demonstra como o algoritmo de transformação foi capaz de modelar um conhecimento embutido no espaço exploratório.

Outro comportamento importante retratado na Figura 4.16 é que conforme β aumenta, mais pares de baixo custo são escolhidos. Quando $\beta = 15$ a quantidade de pares dentro dos intervalos $[0.00, 0.25)$ e $[0.25, 0.50)$ é maior e a quantidade de sequências com pares nos intervalos $[0.50, 0.75)$ e $[0.75, 1.00)$ é menor do que quando $\beta = 1$. Para $\beta = 5$ os valores ficam em uma posição mais intermediária com relação às outras configurações. Esse comportamento era esperado, pois o parâmetro β do *Ant System* corresponde à importância que é dada à visibilidade da formiga, a qual, como mencionado anteriormente, é o inverso da distância do nó atual da formiga com relação ao próximo nó. Quanto maior o valor de β, mais importância é dada a visibilidade e assim, a probabilidade (Expressão 3.12) da formiga escolher o nó de menor custo é maior. Desse modo, para valores altos de β ($\beta = 15$), as soluções geradas no *Ant System* tenderam a possuir arestas (pares) com um custo menor, o que acarretou a geração de sequências de pares de otimizações com custo mais baixo.

A Figura 4.17 retrata esse comportamento com mais clareza. Nela estão os valores do percentual de pares formados com a otimização i que estão nos intervalos $[0.00, 0.25)$ e $[0.25, 0.50)$ ordenados para as otimizações do experimento com $\beta = 1$. Quando $\beta = 1$ é possível notar que a porcentagem de pares situados entre $[0.00, 0.25)$ e $[0.25, 0.50)$ varia bastante, no entanto o mesmo não ocorre quando $\beta = 15$. Nesta configuração, a porcentagem da maioria dos pares varia entre 0.8 e 1.0, de modo que a porcentagem de pares situados nos intervalos $[0.00, 0.25)$ e $[0.25, 0.50)$ é maior do que quando $\beta = 1$. Quando $\beta = 5$ a maioria dos pontos fica entre as duas outras linhas. Esse comportamento indica que aumentando β os pares de menor custo - os melhores pares de acordo com o

algoritmo de transformação - tendem a serem escolhidos pelo *Ant System* e diminuindo o parâmetro da visibilidade a escolha tende a ser mais aleatória.

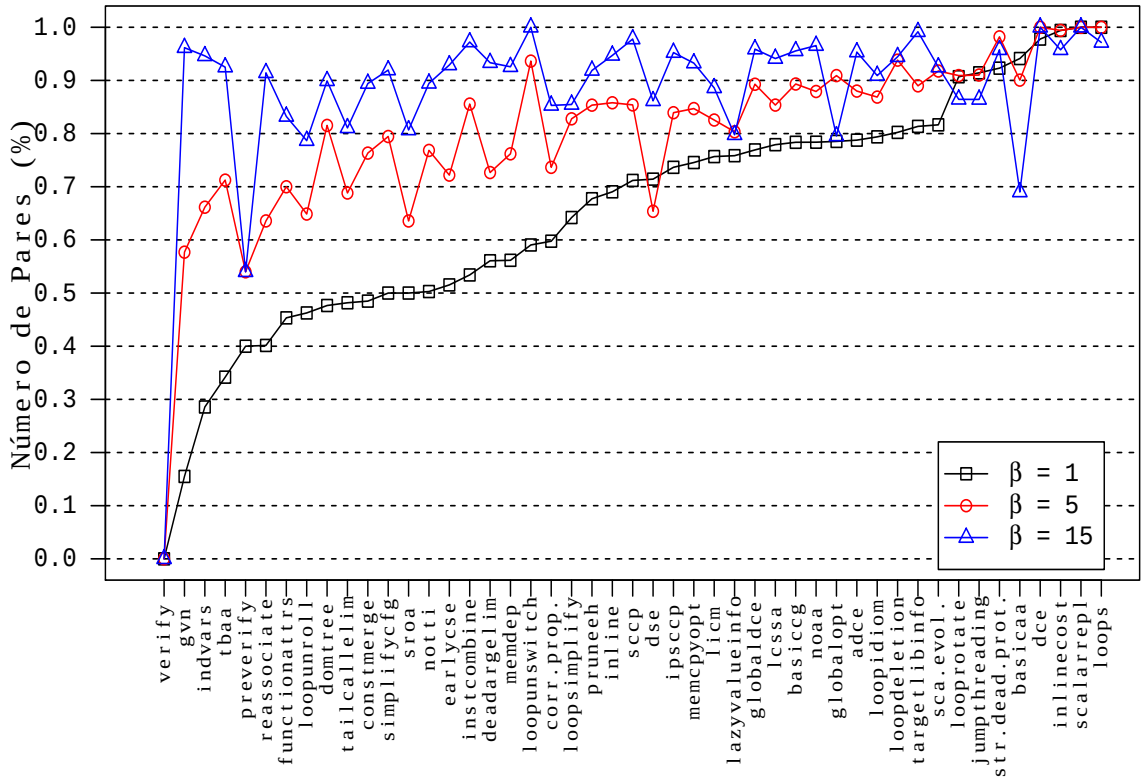
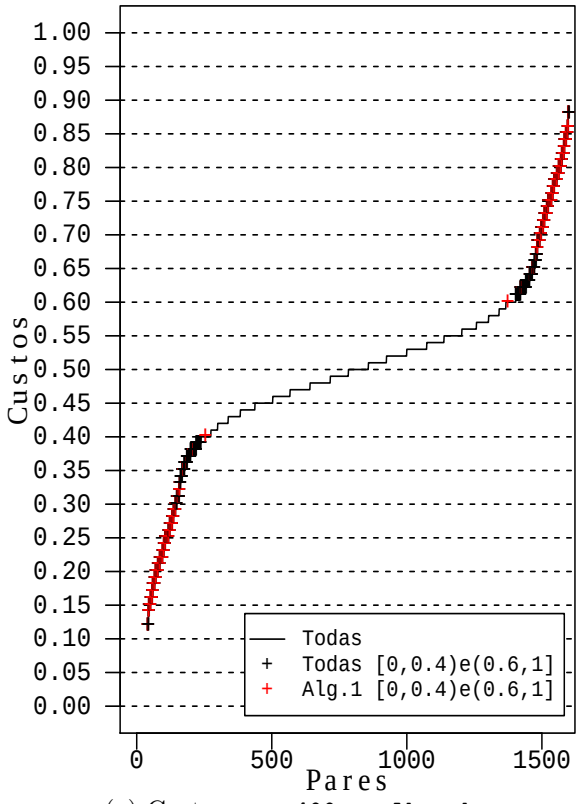


Figura 4.17: Quantidade de pares de otimizações com custo entre $[0.00, 0.25]$ e entre $(0.25, 0.50]$ para $\beta = \{1, 5, 15\}$ em ordem crescente para $\beta = 1$.

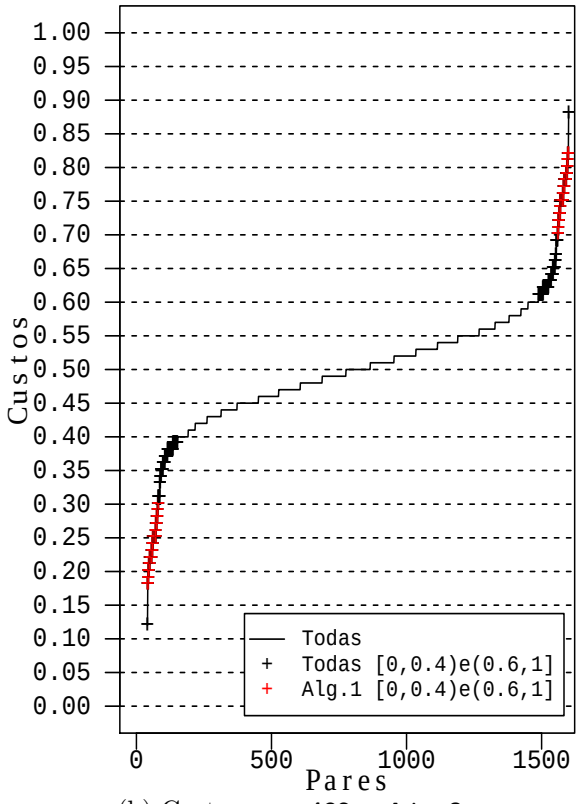
Essa observação é importante pois demonstra a adequação do algoritmo heurístico ao problema e como a sua configuração influência na geração das sequências.

Esses resultados evidenciam que o algoritmo de transformação é capaz de extrair algum conhecimento de *ES* e que a abordagem de resolução do problema com o *Ant System* é capaz de gerar sequências aleatórias, porém enviesadas pelo modelo criado.

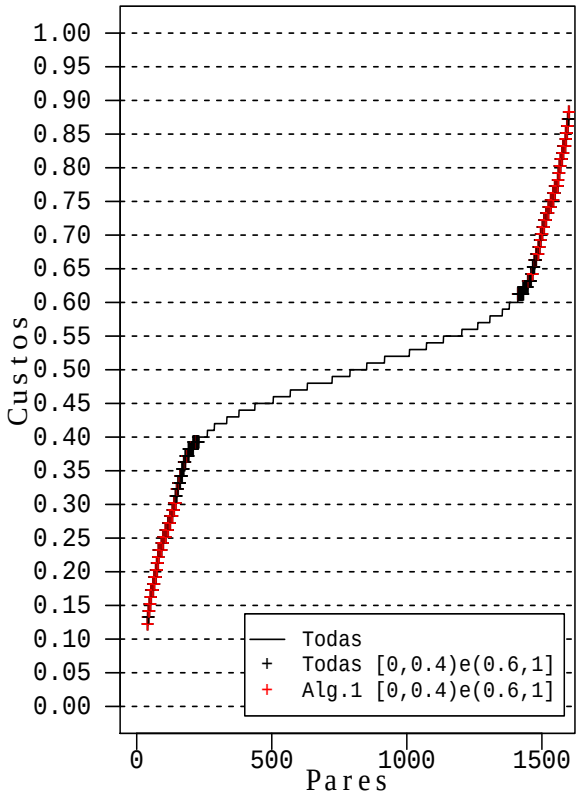
Entretanto, é provável que o conhecimento contido no espaço exploratório não foi suficiente para a obtenção de maiores *speedups*. Para tentar mensurar a quantidade de conhecimento, de acordo com a proposta de frequência de pares de otimizações, presente em *ES* a Figura 4.18 exibe os custos de todos os pares de otimizações para alguns programas - os demais programas seguem um comportamento similar. A linha `Todas` denota o custo para cada par, as cruzes pretas representam apenas os pares de `Todas` que tiveram custo no intervalo $(0.00, 0.40]$ e $(0.60, 1.00]$ e as cruzes vermelhas representam os pares com custo nestes mesmos intervalos, porém que envolveram alguma otimização do Algoritmo 1.



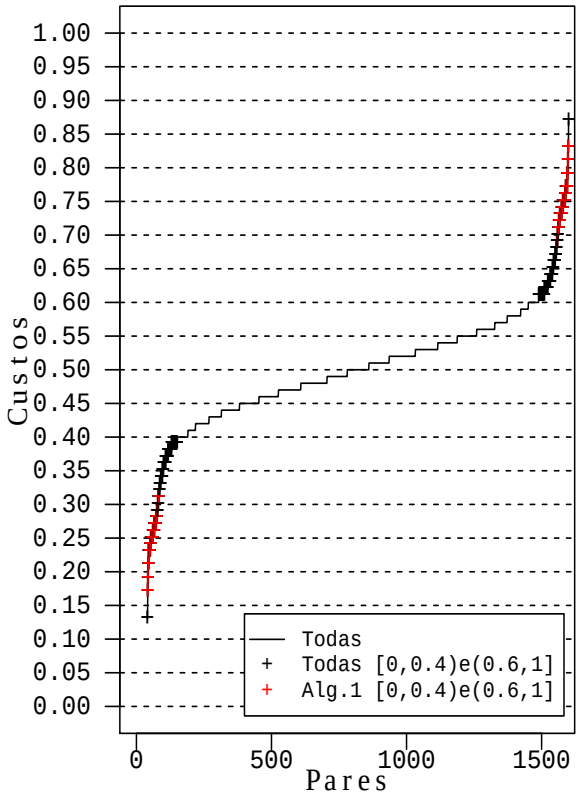
(a) Custos para `400.perlbench`.



(b) Custos para `482.sphinx3`.



(c) Custos para `433.milc`.



(d) Custos para `458.sjeng`.

Figura 4.18: Custos de todos os pares de otimizações em ordem crescente para os programas `400.perlbench`, `482.sphinx3`, `433.milc` e `458.sjeng`.

Na Figura 4.18 nota-se que a maioria dos pares teve um custo intermediário, próximo a 0.50. Como demonstrado na Figura 4.16, o *Ant System* consegue escolher aqueles pares com custo menor do que 0.50, porém se o custo do par fica em uma zona intermediária, como $(0.40, 0.60]$, este valor não é tão relevante como conhecimento do espaço exploratório. Para os programas da Figura 4.18 os pares significativos são aqueles denotados pelas cruzes pretas que estão nos extremos do gráfico e que indicam mais fortemente se o par é bom ou ruim. No entanto, esses pares estão em menor quantidade e ocupam uma pequena faixa de valores. Além disso, dentre os pares nos intervalos $(0.40, 0.60]$ e $(0.60, 1]$ a grande maioria corresponde às otimizações do Algoritmo 1. Este comportamento indica que, desconsiderando o conhecimento proporcionado pelo Algoritmo 1, poucos pares de ES apresentaram uma frequência alta ou baixa, a qual indicasse o benefício de aplicação do par. Este fator é uma hipótese para explicar o aumento de desempenho marginal para os programas, pois demonstra que ES apresentou uma quantidade pequena de pares significativos para o aumento de desempenho das sequências.

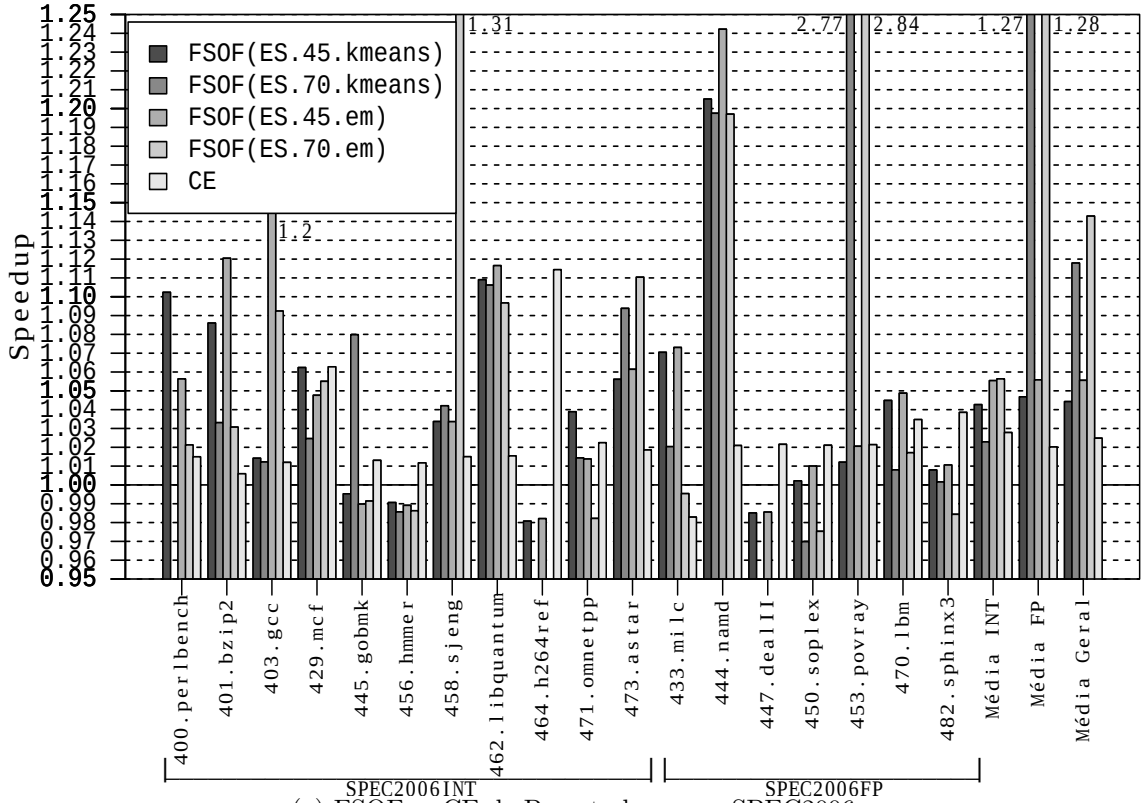
4.4.3 Comparação FSOF

Os resultados de *speedup* do FSOF foram comparados com o algoritmo *Combined Elimination* (CE) proposto por Pan et al. (Pan e Eigenmann, 2006), o qual foi implementado como descrito na Seção 2.4.1.

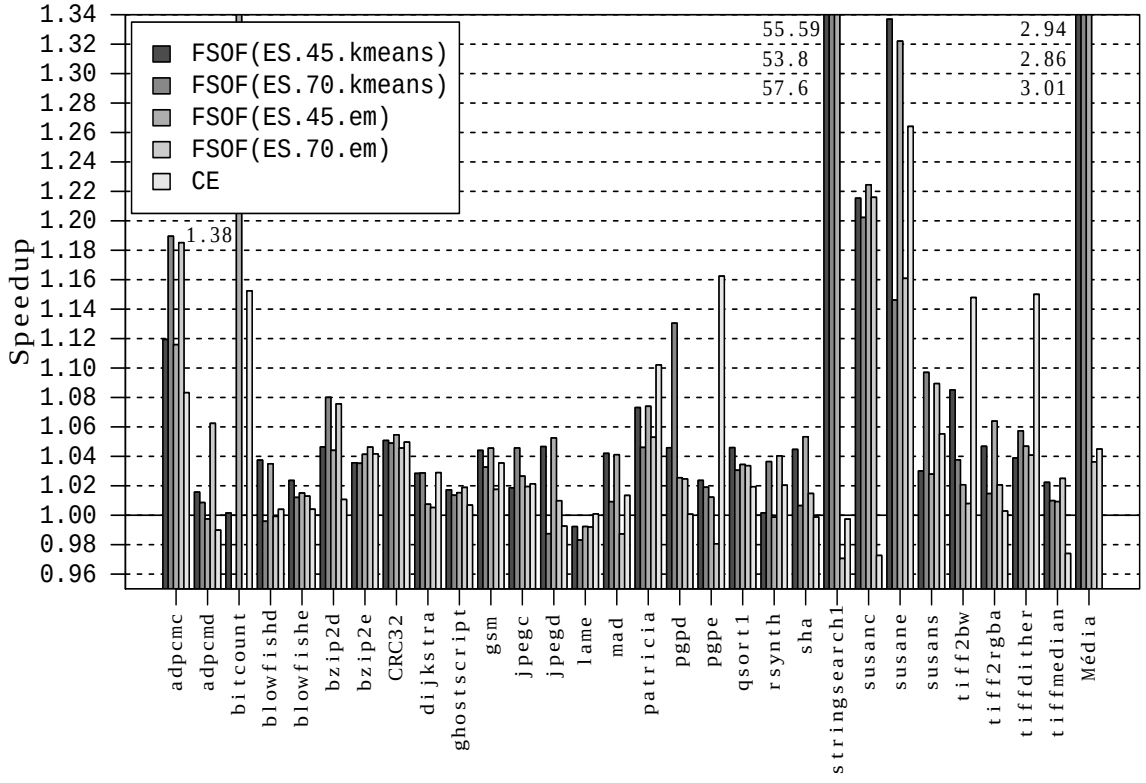
A Figura 4.19(a) apresenta os *speedups* de todas as configurações do FSOF e do CE para o SPEC2006. Dos 18 programas desta suíte de *benchmarks* 13 obtiveram *speedups* maiores do que o CE para alguma estratégia do FSOF. Isoladamente, cada estratégia do *framework* teve desempenho superior do que o CE para pelo menos metade dos *benchmarks*. Além disso, o ganho médio do CE (02.49%) foi inferior ao ganho médio do FSOF para todas as configurações.

Na Figura 4.19(b), com os resultados para o cBench, pode-se notar que 22 programas obtiveram um maior desempenho com o FSOF e cada estratégia sozinha foi melhor do que o CE para mais da metade dos *benchmarks*. Com relação ao ganho médio, apenas a estratégia `ES.70.em` do FSOF teve um valor inferior do que o CE, o qual obteve 04.49% de ganho.

Estes resultados indicam que o FSOF é melhor para a otimização de programas do que o CE. Além disso, o CE necessita de um número muito grande de avaliações para alcançar os valores de *speedup* exibidos. A Figura 4.20 compara a quantidade de vezes que cada programa foi compilado e executado com uma sequência diferente para cada programa do conjunto de teste. Para o SPEC2006 e para o cBench a quantidade de avaliações que o

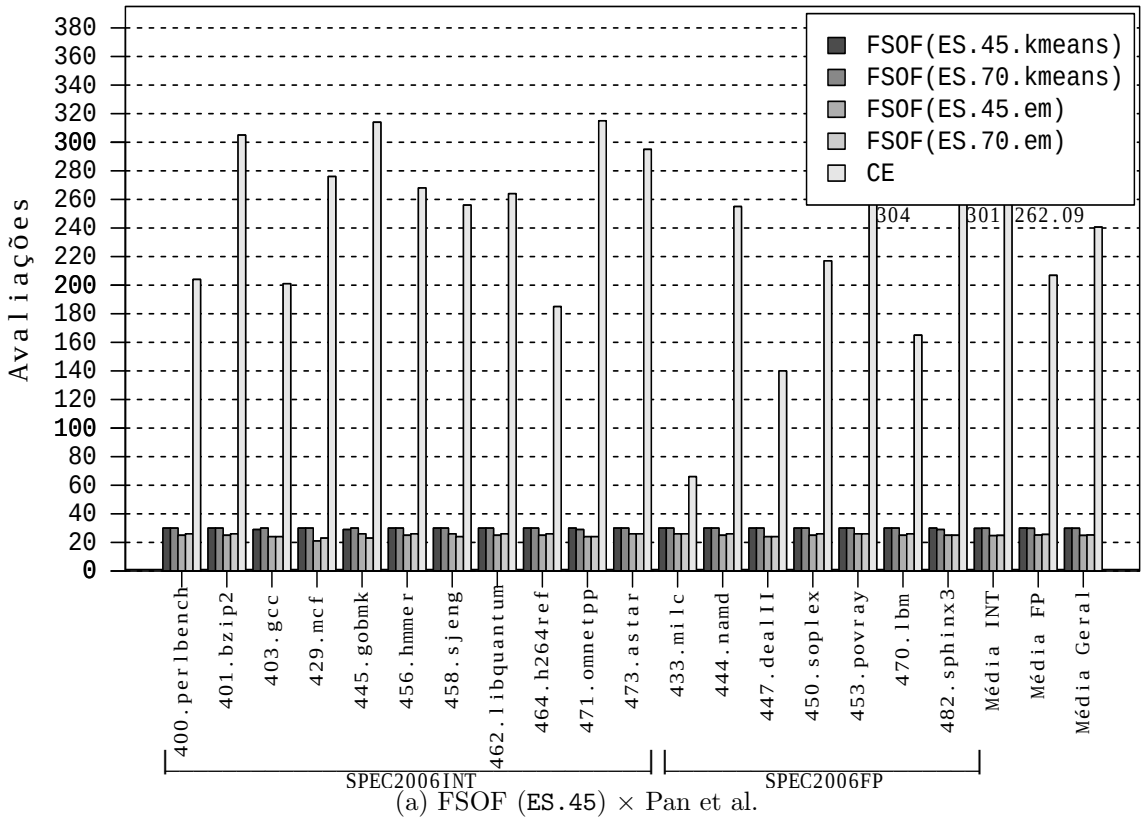


(a) FSOF × CE de Pan et al. para o SPEC2006.

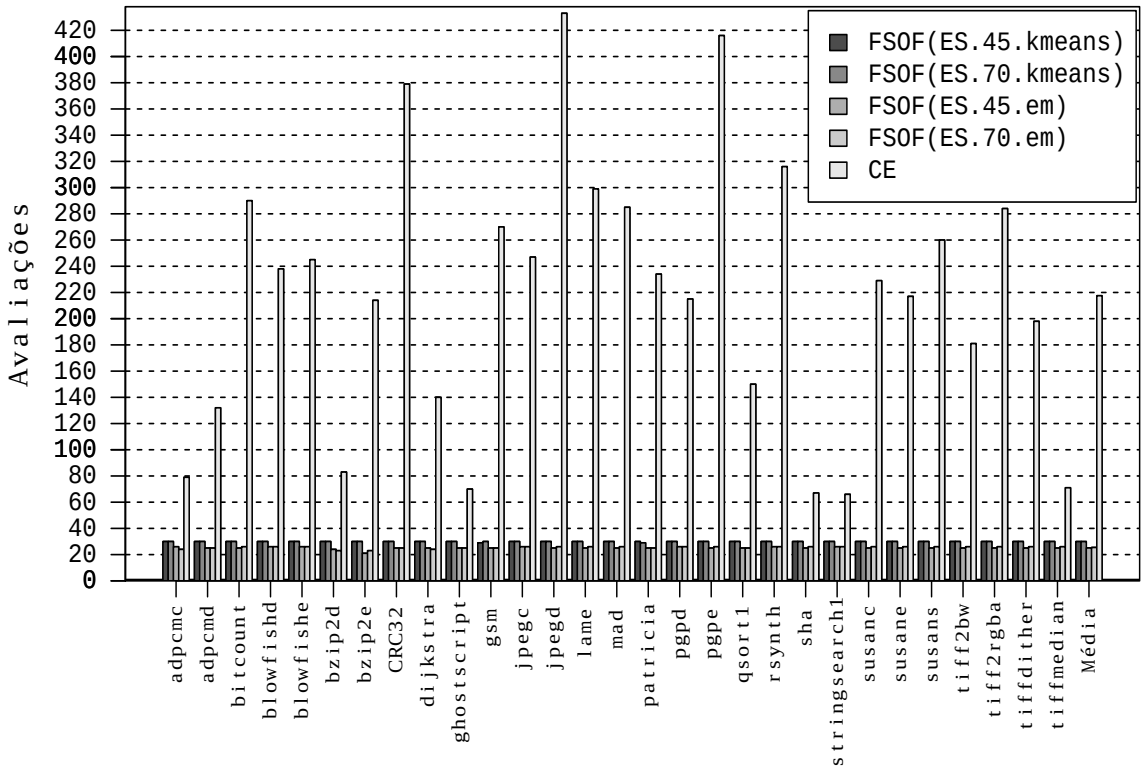


(b) FSOF × CE de Pan et al. para o cBench.

Figura 4.19: Comparação dos *speedups* do FSOF com o *Combined Elimination* (`CE`) de Pan et al. para os *benchmarks* do SPEC2006 e cBench.



(a) FSOF (ES.45) × Pan et al.



(b) FSOF (ES.70) × Pan et al.

Figura 4.20: Comparação do número de avaliações do FSOF com o *Combined Elimination* de Pan et al. para os *benchmarks* do SPEC2006 e cBench.

FSOF precisa realizar é algo em torno de 25 ou 30. Já para o CE os programas com menor número de avaliações (`433.milc` e `stringsearch1`) foram avaliados 66 vezes. Na média, os programas do SPEC2006 necessitaram de 240.61 e do cBench de 217.52 avaliações com CE para apresentar um desempenho inferior ao FSOF.

4.5 Considerações Gerais

Este capítulo apresentou os resultados para as abordagens propostas para mitigação do POF e do PSF. A estratégia de seleção foi capaz de otimizar a maioria dos programas, necessitando de poucas avaliações para isso. A estratégia de ordenação aumentou o desempenho marginalmente e contribuiu para aumentar o desempenho geral do FSOF.

Foi demonstrado que a quantidade de otimizações para gerar as sequências do espaço exploratório não influência significativamente os resultados. Os experimentos indicaram que uma configuração com menos otimizações proporcionou desempenho equivalente a outra com mais transformações.

Na estratégia de seleção foi possível demonstrar que os *speedups* são dependentes e aproximadamente proporcionais ao desempenho obtido pelo mecanismo de conhecimento prévio.

Na estratégia de ordenação algumas evidências foram reunidas que indicam que para a modelagem considerando a frequência das otimizações, o algoritmo de transformação proposto é capaz de extrair conhecimento do espaço exploratório. Além disso, o algoritmo de colônia de formigas se mostrou adequado para induzir a busca de sequências.

5

Conclusões e Trabalhos Futuros

Os problemas de seleção e ordenação de fase são duas questões da área de otimização de compiladores que ainda não possuem soluções efetivas. O principal desafio consiste em descobrir no imenso espaço de busca a melhor sequência de otimizações, na melhor ordem, necessitando da menor quantidade de avaliações possíveis.

Os melhores resultados da literatura foram alcançados principalmente com estratégias de compilação iterativa, que necessita de um número excessivo de avaliações. A adoção de aprendizagem de máquina e a ideia de armazenar conhecimento prévio para prever novas sequências tem diminuído a quantidade de avaliações, porém com desempenho nem sempre como esperado.

Neste contexto, este trabalho apresentou duas abordagens para mitigação do PSF e do POF, as quais são integradas para fornecer sequências de alto desempenho com relativamente poucas avaliações. As duas propostas foram implementadas em um *framework* de otimização de programas denominado FSOF que integra as estratégias e faz uso de uma estrutura de conhecimento prévio de programas e sequências.

A estratégia de seleção foi capaz de aumentar o desempenho para a maioria dos programas e para alguns deles, conseguiu encontrar sequências que diminuíram substancialmente o tempo de execução. A estratégia de ordenação teve desempenho comparável à sequência fixa do compilador, mostrou ser viável em um cenário em que o tempo de compilação não é um fator crítico e possibilitou o aumento do desempenho geral do FSOF.

Os *speedups* alcançados com o FSOF foram superiores ao *Combined Elimination*, uma estratégia na literatura que apresentou bons resultados. No entanto, o FSOF necessitou de muito menos avaliações dos programas demonstrando sua efetividade para otimizar programas.

Analisando a estrutura de conhecimento prévio foi possível concluir que a quantidade de otimizações não é um fator determinante na qualidade das sequências geradas. Em experimentos realizados com quantidades diferentes de otimizações, ficou evidenciado que mesmo com uma quantidade menor de otimizações para formar as sequências, se elas forem significativas - como aquelas que compõem a sequência padrão do compilador - os resultados são semelhantes àqueles com sequências formadas com mais otimizações.

Na estratégia de seleção, a atuação do algoritmo de clusterização mostrou ser importante, pois foi demonstrado que entre dois algoritmos avaliados, um deles, por agrupar apenas os programas mais semelhantes, conseguiu obter desempenho equivalente ao outro. Este agrupamento proporcionou a otimização dos programas com um número menor de avaliações.

Com relação à abordagem de ordenação, o algoritmo de transformação foi capaz de gerar um modelo do POF que considera a frequência de pares de otimizações das sequências do espaço exploratório. Além disso, o modelo conseguiu refletir algum conhecimento que estava presente no mecanismo de conhecimento prévio. Foi demonstrado também que o algoritmo de colônia de formigas é adequado para a modelagem proposta.

5.1 Trabalhos futuros

Algumas questões podem ser melhor exploradas em trabalhos futuros, tanto para propor estratégias que simplesmente buscam aumentar o desempenho de programas, como para explorar o difícil relacionamento existente entre os programas e as sequências de otimizações. Algumas sugestões são:

Metodologia de construção do espaço exploratório Os espaços exploratórios deste trabalho foram construídos por meio de algoritmo aleatório de compilação iterativa. Outras formas de construir este mecanismo de conhecimento prévio como algoritmos genéticos podem ser experimentadas. Como demonstrado, a qualidade dos resultados de uma estratégia de seleção que faz uso de uma estrutura como essa tem seus resultados em função da qualidade das sequências presentes no mecanismo de conhecimento prévio.

Avaliar outras formas de caracterização dos programas Utilizar novas estratégias de caracterização de programas, as quais permitam uma maior correlação entre os *speedups* obtidos e a semelhança entre os programas.

Adicionar conhecimento no espaço exploratório A fim de tentar aumentar o desempenho para o MOF, a construção do espaço exploratório poderia ser investigada com mecanismos que fazem uso mais agressivo do conhecimento dos projetistas do compilador. Como o algoritmo proposto mostrou ser capaz de modelar informações presentes no espaço exploratório, caso se consiga reunir sequências que possuam pares de otimizações suficientemente significativos é possível que o desempenho possa ser melhorado.

Maior experimentação do MSF Principalmente em novos trabalhos que façam uso de mais programas, é possível experimentar outros algoritmos de clusterização para realizar os agrupamentos. Pelo fato dos algoritmos avaliados terem determinado a quantidade avaliações, a tentativa de novas estratégias pode fazer com que os mesmos resultados possam ser alcançados com menos avaliações.

Maior experimentação do MOF Existe a possibilidade de explorar mais o MOF durante a transformação do POF para PCVA. Por exemplo, ao invés de selecionar todas as sequências do espaço exploratório, uma abordagem seria escolher apenas as k melhores sequências de cada programa de ES. Desse modo, haveria a possibilidade de avaliar se, considerando apenas um pequeno conjunto das sequências mais relevantes de cada programa, o desempenho aumentaria.

5.2 Considerações Finais

Os *speedups* alcançados pelo FSOF foram significativos e superaram a abordagem da literatura implementada para comparação. Ressalta-se que os *benchmarks* empregados nos experimentos são alguns dos mais representativos considerando aplicações científicas e de sistemas embarcados. Estes resultados foram conseguidos com relativamente poucas avaliações dos programas.

A avaliação experimental também possibilitou uma análise ampla sobre algumas questões relevantes, as quais são concernentes à área de otimização em compiladores: quantidade de otimizações para geração das sequências, influência do espaço exploratório na estratégia de seleção e possibilidade de extrair conhecimento do mecanismo de conhecimento prévio a partir de informações das sequências.

REFERÊNCIAS

Agakov, F.; Bonilla, E.; Cavazos, J.; Franke, B.; Fursin, G.; O'Boyle, M. F. P.; Thomson, J.; Toussaint, M.; Williams, C. K. I. Using machine learning to focus iterative optimization. In: *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2006, p. 295–305.

Barreteau, M.; Bodin, F.; Chamski, Z.; Charles, H.-P.; Eisenbeis, C.; Gurd, J.; Hoogerbrugge, J.; Hu, P.; Jalby, W.; Kisuki, T.; Knijnenburg, P.; Mark, P.; Nisbet, A.; O'Boyle, M.; Rohou, E.; Seznec, A.; Stöhr, E.; Treffers, M.; Wijshoff, H. Oceans - optimising compilers for embedded applications. In: *Parallel Processing*, Springer Berlin Heidelberg, 1999, p. 1171–1175 (*Lecture Notes in Computer Science*, v.1685).

Cavazos, J.; Fursin, G.; Agakov, F.; Bonilla, E.; O'Boyle, M. F. P.; Temam, O. Rapidly selecting good compiler optimizations using performance counters. In: *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2007, p. 185–197.

Cavazos, J.; O'Boyle, M. F. P. Method-specific dynamic compilation using logistic regression. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA: ACM, 2006, p. 229–240.

cBench. The collective benchmarks. Data de acesso: 02 de Março de 2014, 2013. Disponível em `http://ctuning.org/wiki/index.php/CTools:CBench`

Chabbi, M. M.; Mellor-Crummey, J. M.; Cooper, K. D. Efficiently exploring compiler optimization sequences with pairwise pruning. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, New York, NY, USA: ACM, 2011, p. 34–45.

Che, Y.; Wang, Z. A lightweight iterative compilation approach for optimization parameter selection. In: *First International Multi-Symposiums on Computer and Computational Sciences*, 2006, p. 318–325.

Cooper, K. D.; Grosul, A.; Harvey, T. J.; Reeves, S.; Subramanian, D.; Torczon, L.; Waterman, T. Exploring the structure of the space of compilation sequences using randomized search algorithms. *Journal of Supercomputing*, v. 36, n. 2, p. 135–151, 2006.

Cooper, K. D.; Schielke, P. J.; Subramanian, D. Optimizing for reduced code space using genetic algorithms. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA: ACM, 1999, p. 1–9.

Cooper, K. D.; Subramanian, D.; Torczon, L. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, v. 23, n. 1, p. 7–22, 2002.

Cooper, K. D.; Waterman, T. Investigating adaptive compilation using the mipspro compiler. In: *Proceedings of the Symposium of the Los Alamos Computer Science Institute*, 2003.

Dempster, A.; Laird, N.; Rubin, D. Maximum likelihood from incomplete data via the em algorithm. *J. Royal Statistical Society, Series B*, v. 39, n. 1, p. 1–38, 1977.

Dorigo, M.; Maniezzo, V.; Colorni, A. Ant system: Optimization by a colony of cooperating agents. *Transactions on Systems, Man and Cybernetics*, v. 26, n. 1, p. 29–41, 1996.

Fursin, G.; Cavazos, J.; O'Boyle, M.; Temam, O. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In: *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers*, 2007.

Fursin, G. G.; O'Boyle, M. F. P.; Knijnenburg, P. M. W. Evaluating iterative compilation. In: *Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer-Verlag, 2005, p. 362–376.

Gheorghita, S. V.; Corporaal, H.; Basten, T. Iterative compilation for energy reduction. *Journal of Embedded Computing*, v. 1, n. 4, p. 509–520, 2005.

Guthaus, M. R.; Ringenberg, J. S.; Ernst, D.; Austin, T. M.; Mudge, T.; Brown, R. B. Mibench: A free, commercially representative embedded benchmark suite. In: *Proceedings of IEEE International Workshop on Workload Characterization*, Washington, DC, USA: IEEE Computer Society, 2001, p. 3–14.

Haneda, M.; Knijnenburg, P. M. W.; Wijshoff, H. A. G. Generating new general compiler optimization settings. In: *Proceedings of the 19th Annual International Conference on Supercomputing*, New York, NY, USA: ACM, 2005, p. 161–168.

Hartigan, J. A. *Clustering algorithms*. 99th ed. New York, NY, USA: John Wiley & Sons, Inc., 1975.

Henning, J. L. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, v. 34, n. 4, p. 1–17, 2006.

Hintze, J. L.; Nelson, R. D. Violin plots: A box plot-density trace synergism. *The American Statistician*, v. 52, 1998.

Hoste, K.; Eeckhout, L. Cole: Compiler optimization level exploration. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, New York, NY, USA: ACM, 2008, p. 165–174.

Jackson, J. *A user's guide to principal components*. Wiley series in probability and mathematical statistics. New York [u.a.]: Wiley, 1991.

Jantz, M. R.; Kulkarni, P. A. Performance potential of optimization phase selection during dynamic jit compilation. In: *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA: ACM, 2013, p. 131–142.

Kisuki, T.; Knijnenburg, P.; O'Boyle, M.; Wijshoff, H. A. G. Iterative compilation in program optimization. 2000a.

Kisuki, T.; Knijnenburg, P. M. W.; O'Boyle, M. Combined selection of tile sizes and unroll factors using iterative compilation. In: *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2000b, p. 237–246.

Kisuki, T.; Knijnenburg, P. M. W.; O'Boyle, M. F. P.; Bodin, F.; Wijshoff, H. A. G. A feasibility study in iterative compilation. In: *Proceedings of the Second International Symposium on High Performance Computing*, London, UK, UK: Springer-Verlag, 1999, p. 121–132.

KUFRIN, R. Perfsuite: An accessible, open source performance analysis environment for linux. In: *Proceedings of the Linux Cluster Conference, Chapel*, 2005.

KULKARNI, P.; HINES, S.; HISER, J.; WHALLEY, D.; DAVIDSON, J.; JONES, D. Fast searches for effective optimization phase sequences. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, 2004, p. 171–182.

KULKARNI, P. A.; HINES, S. R.; WHALLEY, D. B.; HISER, J. D.; DAVIDSON, J. W.; JONES, D. L. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization*, v. 2, n. 2, p. 165–198, 2005.

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

KULKARNI, P. A.; WHALLEY, D. B.; TYSON, G. S.; DAVIDSON, J. W. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization*, v. 6, n. 1, p. 1:1–1:36, 2009.

KULKARNI, P. P. Fast and effective solutions to the phase ordering problem. In: *Eletronic Theses, Treatises and Dissertions*, 2007.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, Washington, DC, USA: IEEE Computer Society, 2004, p. 75–.

LEATHER, H.; O'BOYLE, M.; WORTON, B. Raced profiles: Efficient selection of competing compiler optimizations. In: *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA: ACM, 2009, p. 50–59.

LOKUCIEJEWSKI, P.; PLAZAR, S.; FALK, H.; MARWEDEL, P.; THIELE, L. Approximating pareto optimal compiler optimization sequences - a trade-off between wcet, acet and code size. *Journal of Software Practice Experience*, v. 41, n. 12, p. 1437–1458, 2011.

LONG, S.; FURSIN, G. A heuristic search algorithm based on unified transformation framework. In: *International Conference Workshops on Parallel Processing*, 2005, p. 137–144.

Long, S.; O'Boyle, M. Adaptive java optimisation using instance-based learning. In: *Proceedings of the 18th Annual International Conference on Supercomputing*, New York, NY, USA: ACM, 2004, p. 237–246.

Malik, A. M. Spatial based feature generation for machine learning based optimization compilation. In: *Proceedings of the Ninth International Conference on Machine Learning and Applications*, Washington, DC, USA: IEEE Computer Society, 2010, p. 925–930.

Mucci, P. J.; Browne, S.; Deane, C.; Ho, G. Papi: A portable interface to hardware performance counters. In: *Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, p. 7–10.

Muchnick, S. S. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

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.; Cavazos, J.; Alvarez, M. A. Using graph-based program characterization for predictive modeling. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, New York, NY, USA: ACM, 2012, p. 196–206.

Park, E.; Kulkarni, S.; Cavazos, J. An evaluation of different modeling techniques for iterative compilation. In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, New York, NY, USA: ACM, 2011, p. 65–74.

Polybench. The polyhedral benchmark suite. Data de acesso: 02 de Março de 2014, 2013.
Disponível em `http://www.cs.ucla.edu/~pouchet/software/polybench/`

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.

Siddheswar Ray, S. R. Determination of number of clusters in k-means clustering and application in colour image segmentation. In: *Proceedings of the 4th International Conference on Advances in Pattern Recognition and Digital Techniques*, 1999.

Srikant, Y. N.; Shankar, P. *The compiler design handbook: Optimizations and machine code generation*. 2nd ed. Boca Raton, FL, USA: CRC Press, Inc., 2007.

Thomson, J.; O'Boyle, M.; Fursin, G.; Franke, B. Reducing training time in a one-shot machine learning-based compiler. In: *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer-Verlag, 2010, p. 399–407.

WEKA Waikato environment for knowledge analysis. Data de acesso: 02 de Março de 2014, 2013.
Disponível em `http://www.cs.waikato.ac.nz/ml/weka/`

Whitfield, D. L.; Soffa, M. L. An approach for exploring code improving transformations. *ACM Transactions on Programming Language System*, v. 19, n. 6, p. 1053–1084, 1997.

Zhao, W.; Cai, B.; Whalley, D.; Bailey, M. W.; van Engelen, R.; Yuan, X.; Hiser, J. D.; Davidson, J. W.; Gallivan, K.; Jones, D. L. Vista: A system for interactive code improvement. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers For Embedded Systems*, New York, NY, USA: ACM, 2002, p. 155–164.

Zhou, Y.-Q.; Lin, N.-W. A study on optimizing execution time and code size in iterative compilation. In: *Third International Conference on Innovations in Bio-Inspired Computing and Applications*, 2012, p. 104–109.