

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Tiago Pereira de Faria

**Uso de Algoritmo baseado em colônia de
formigas para explorar sequências de otimização
do compilador**

Uberlândia, Brasil

2019

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Tiago Pereira de Faria

**Uso de Algoritmo baseado em colônia de formigas para
explorar sequências de otimização do compilador**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do grau
de Bacharel em Ciência da Computação.

Orientador: Luiz Gustavo Almeida Martins

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2019

Tiago Pereira de Faria

Uso de Algoritmo baseado em colônia de formigas para explorar sequências de otimização do compilador

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 12 de julho de 2019:

Luiz Gustavo Almeida Martins
Orientador

Carlos Roberto Lopes

Murillo Guimarães Carneiro

Uberlândia, Brasil
2019

Resumo

Achar sequências de passos de otimização específicas para o código alvo é uma tarefa complicada, porém muito importante, pois conseguem melhorar o desempenho significativamente em relação às sequências pré estabelecidas tipicamente presentes nos compiladores modernos. Este trabalho propõem um modelo híbrido utilizando dois módulos principais: um seletor e um ordenador. O seletor visa selecionar, dentre um conjunto de códigos de referência, aqueles que com maior similaridade com o novo código. Para isso, foram avaliadas duas abordagens, uma baseada no KNN e outra utilizando um algoritmo de K-Medias (*K-Means*). O modulo ordenador tem como objetivo encontrar a melhor ordem de aplicação dos passos presentes nas sequências dos códigos similares selecionados. Para ele, foram avaliadas duas abordagens diferentes utilizando algoritmos baseados em colônia de formigas. A primeira utiliza a distância percorrida no grafo para avaliar as soluções das formigas, enquanto a segunda compila e simula a execução do código alvo para avalia-las. Foram realizados experimentos utilizando 51 programas do *benchmark* do *Test-Suite* do compilador LLVM. Com os experimentos foi possível concluir que o KNN consegue selecionar de forma mais eficiente os programas similares. O modelo utilizando a distância percorrida no grafo para avaliar as sequências apresentou *speedup* médio de 1.05x em relação ao melhor nível de otimização (-OX) do compilador LLVM. O modelo utilizando simulação das sequências conseguiu *speedup* de 1.073x em relação ao -OX, embora demande um tempo de exploração consideravelmente maior (91.1 minutos ao invés de 2.7).

Palavras-chave: Compiladores, Algoritmo Baseado em colônia de formigas, ACO, Busca de sequências de passos de otimização

Lista de ilustrações

Figura 1 – Funcionamento do algoritmo KNN	17
Figura 2 – Funcionamento do algoritmo K-Means	18
Figura 3 – Comportamento de formigas em procura alimento	19
Figura 4 – Funcionamento básico de um algoritmo baseado em colônia de formigas	20
Figura 5 – Esquematização do funcionamento das fases de preparação e entrega do sistema de otimização automático de código	24
Figura 6 – Exemplo de grafo de frequência de passos de otimização	28
Figura 7 – Relação entre o custo e a latência de soluções avaliadas para o código <i>treesort</i> do <i>Benchmark</i> do LLVM	30
Figura 8 – Comparação entre K-Means e KNN para selecionar programas similares	33

Lista de tabelas

Tabela 1 – Características Estáticas, retiradas de (JUNIOR, 2016)	25
Tabela 2 – Programas do Benchmark	31
Tabela 3 – Resultado dos experimentos preliminares com o algoritmo KNN	33
Tabela 4 – Resultados dos experimentos preliminares com o algoritmo <i>K-Means</i> .	34
Tabela 5 – Resultados das execuções do ACO com grafo de frequência	35
Tabela 6 – Resultados das execuções do ACO com ambos os tipos de avaliação e do KNN	36

Sumário

1	INTRODUÇÃO	8
1.1	Justificativa	11
1.2	Objetivos	12
1.3	Organização do texto	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Compiladores e otimização	13
2.2	Problema de Seleção e Ordenação de Otimizações	14
2.2.1	Decidibilidade do problema	15
2.2.2	O Espaço de Busca	16
2.2.3	Formas de redução do espaço de busca	16
2.2.4	Otimização por Colônia de Formigas	19
2.3	Trabalhos Relacionados	21
3	DESENVOLVIMENTO	24
3.1	Extração de características	25
3.2	Seleção de programas similares	27
3.3	Construção do grafo	28
3.4	Ordenando as sequências selecionadas	29
4	EXPERIMENTOS	31
4.1	Ambiente de Experimentação	31
4.2	Análise das abordagens de seleção de programas similares	32
4.3	Análise das estratégias de avaliação usadas no ACO	35
5	CONCLUSÃO	37
	REFERÊNCIAS	39
	APÊNDICES	41
	APÊNDICE A – RESULTADOS DO MODELO USANDO KNN (K = 6) E AVALIAÇÃO POR GRAFO DE FREQUÊNCIAS	42

APÊNDICE B – RESULTADOS DO MODELO USANDO KNN ($K = 9$) E AVALIAÇÃO POR GRAFO DE FREQUÊNCIAS	44
APÊNDICE C – RESULTADOS DO MODELO USANDO KNN ($K = 12$) E AVALIAÇÃO POR GRAFO DE FREQUÊNCIAS	46
APÊNDICE D – RESULTADOS DO MODELO USANDO KNN($K = 6$) E AVALIAÇÃO POR COMPILAÇÃO DO CÓDIGO	48
APÊNDICE E – RESULTADOS DO MODULO SELETOR BASEADO NO <i>KNN</i> PARA DIFERENTES VALORES DE K	50

1 Introdução

Um compilador tem por função principal converter um código de uma linguagem fonte para uma linguagem alvo (AHO; SETHI; LAM, 2008), comumente usado para transformar programas escritos em linguagens de alto nível para uma linguagem capaz de ser executada por um computador, um código de máquina. A maioria dos compiladores também possui uma fase de otimização do código, que atua em um código intermediário gerado pelo compilador, que pode melhorar o desempenho do código final em diversas métricas, como tempo de execução, gasto energético e espaço de memória usado. Nesta etapa, existem diversas análises e transformações, conhecidas como passos de otimização, que podem ser aplicadas, sendo possível optar ou não por usar cada uma delas. Além disso, em alguns compiladores ainda é possível escolher a ordem em que esses passos serão aplicados. A escolha de quais passos de otimização aplicar e sua ordem tem grande influência no desempenho do código compilado (MARTINS et al., 2016).

A grande quantidade de transformações e a opção de se ajustar a ordem em que são aplicadas, faz com que a tarefa da escolha da sequência de aplicação dos passos de otimização seja um problema complexo. Uma boa sequência pode melhorar o código final compilado e, conseqüentemente, seu desempenho em execução. Por outro lado, uma sequência má formulada pode até mesmo não ser possível de ser aplicada pelo compilador; gerar um código inválido, cujo resultado difere do esperado; ou ainda, podem piorar o desempenho do código executável. Para abstrair esse problema, e possibilitar que o usuário possa usar o compilador e ter um código final otimizado, sem precisar estudar e/ou entender como escolher os passos de otimização a serem usados, grande parte dos compiladores oferece níveis de otimização, que são sequências fixas pré-definidas que podem ser escolhidas por meio das opções -O1, -O2 e -O3 (ASHOURI et al., 2018). Essas sequências são escolhidas pelo desenvolvedor do compilador, com base em sua *expertise*, e visam atender/otimizar o maior número possível de códigos (maior representatividade). Porém, não é garantido que alguma delas seja a sequência ótima ou que sequer sejam benéficas para o desempenho de todo código (COOPER et al., 2006). Em alguns compiladores, o desenvolvedor do software pode escolher manualmente a sequências de passos de otimização que será usada na compilação. Um programador pode conseguir achar uma sequência que melhore o desempenho do código de forma satisfatória, mas esta tarefa requer tempo e depende da experiência do desenvolvedor.

Neste contexto, surgiu o Problema de Seleção e Ordenação de Passos Otimizações (*Phase ordering problem*), que trata da escolha automática de uma sequência específica para o código que está sendo compilado. O conjunto de todas as sequências de otimização possíveis cresce exponencialmente em função da quantidade de otimizações diferentes.

Assim, não é viável utilizar um algoritmo de busca que avalie todas as sequências possíveis (busca exaustiva) (MARTINS, 2016).

Desta forma, faz-se necessário o uso de algoritmos que guiem esta busca, visando obter o melhor desempenho possível para o código compilado. Na literatura, diferentes abordagens têm sido propostas para a exploração de sequências específicas para os códigos compilados (ASHOURI et al., 2018). Essas abordagens podem ser divididas em três categorias: compilação iterativa, compilação preditiva e abordagens híbridas.

Na compilação iterativa, o código é compilado e avaliado diversas vezes aplicando um algoritmo de busca de sequências guiado por uma heurística. Depois de um número arbitrário de iterações, ou de um tempo máximo estipulado, a melhor sequência encontrada na busca é usada (JUNIOR, 2016). Trabalhos recentes têm obtido bons resultados utilizando essa técnica (ASHOURI et al., 2018), porém, compilar o código e avaliar uma sequência leva tempo, e como isso deve ser feito diversas vezes durante a execução do algoritmo, o método possui um alto custo computacional. Desta forma, o tempo de execução deste método depende diretamente da eficiência de como é feita a avaliação de uma sequência de passos. Esta avaliação pode ser feita por um modelo preditivo, que sem precisar executar o código, estima seu desempenho. Este método é mais rápido, porém, menos preciso que executar o código, podendo impactar no desempenho final do algoritmo de busca. Também pode-se utilizar um simulador da arquitetura alvo para avaliar o programa compilado com a sequência. Esse método tem maior precisão que a alternativa preditiva e tem a vantagem de ser possível analisar o comportamento e situações adversas que possam acontecer em tempo de execução, porém possui um custo computacional maior. A forma mais precisa é avaliar a execução do código diretamente na arquitetura alvo, porém, é a que pode levar mais tempo, dependendo da arquitetura. Se for necessário transferir o executável para uma máquina externa, como em um sistema embarcado por exemplo, isso pode ser um gargalo no algoritmo (MARTINS, 2016).

A compilação preditiva baseia-se na criação de um modelo de conhecimento, capaz de mapear relações entre as boas sequências de otimizações e as características de seus códigos. Uma vez treinado, esses modelos são usados no momento da compilação para extrair as características do código e associa-las a uma boa sequência para o código em análise. Isso resolve o problema do alto tempo de resposta, já que sua parte mais custosa é a fase de treino, onde é usada a compilação iterativa para encontrar as melhores sequências de diversos programas (JUNIOR, 2016). Contudo, a eficiência das sequências encontradas pelos métodos preditivos são tipicamente piores que as obtidas por compilação iterativa (ASHOURI et al., 2014). Um problema que é encontrado na maioria das técnicas deste tipo é a escolha das características usadas para a caracterização do código. Existem duas principais formas de se realizar essa caracterização. Uma delas é a forma estática, que usa informações do código fonte, sem a necessidade de se executar o programa. Pode

ser usado, por exemplo, o grafo de fluxo de controle, o qual permite obter informações como quantidade de instruções de cada tipo, número de nós e arestas. Outra maneira de se caracterizar um código é de forma dinâmica, que é feita enquanto o código está sendo executado. Um exemplo de características dinâmicas são os *performance counters*, que são dados relativos ao desempenho do compilador, como número de instruções completadas e quantidade de acessos ao *cache* (XAVIER; SILVA, 2018).

A abordagem híbrida consiste em mesclar técnicas das abordagens anteriores, com o objetivo de usar as vantagens de cada uma e mitigar seus problemas. Por exemplo, pode-se utilizar um algoritmo de aprendizado de máquina para gerar a população inicial que será utilizada em uma técnica de compilação iterativa. O motivo disso é que os resultados obtidos pelo aprendizado de máquina, apesar de serem gerados muito rápido, não alcançam a eficiência das sequências apontadas pelo método iterativo. E ao começar o método iterativo com soluções semi-ótimas, providas da técnica de aprendizado de máquina, são necessários menos iterações para se chegar no resultado ótimo, reduzindo assim o tempo de resposta da exploração.

Este trabalho implementa o modelo híbrido proposto por Xavier (XAVIER; SILVA, 2018), que usa um modelo preditivo, baseado em agrupamento para selecionar os passos de otimização e, então, resolve o problema de ordenação usando o Algoritmo de Colônia de Formigas (*ACO - Ant Colony Optimization*). Em uma fase prévia, de treinamento, um conjunto de referência é formado a partir de códigos previamente compilados. Para cada código desse conjunto, sabe-se a melhor sequência de otimização e seu vetor de características estáticas. Na fase de operação, o mesmo vetor de características é extraído do programa a ser compilado e utilizado para agrupá-lo com os códigos do conjunto de referência. Os códigos de referência que pertencem ao mesmo grupo que o programa de alvo, são considerados similares e, portanto, considera-se que as boas sequências encontradas para eles também são capazes de melhorar o desempenho do código alvo. No entanto, ainda é feita a reordenação dos seus passos de ordenação. Para cada sequência de otimização selecionada no agrupamento, é aplicado o ACO para encontrar a melhor ordem de aplicação dos seus passos de otimização.

O ACO é um método iterativo, que necessita de uma integração com o ambiente de compilação, para que consiga-se avaliar as sequências de otimização e obter as medidas de desempenho do código otimizado. Neste sentido, implementamos nosso modelo no ambiente *LaraD*. Esse ambiente integra a linguagem baseada em aspectos *Lara* com o ambiente de compilação *LLVM* (*Low Level Virtual Machine*) (LLVM, 2019) e o simulador para o processador *Leon3* (GAISLER, 2019a) (usado em sistemas embarcados em *FPGA - Field Programmable Gate Array* (BROWN et al., 2012)), possibilitando compilar e avaliar o programa com as sequências geradas. O algoritmo de colônia de formigas foi implementado de forma que as formigas percorrem um grafo em que cada nó representa

um passo de otimização, e a distância entre um nó e outro é inversamente proporcional a frequência em que eles aparecem nesta ordem nas sequências dos códigos similares.

Desta forma, temos um método rápido, por usar sequências pré-definidas, mas que são ordenadas especificamente para cada programa alvo, de forma iterativa. Nos experimentos, foram analisadas diferentes abordagens para seleção de códigos similares e para a avaliação das soluções geradas pelo *ACO*. Nessa análise comparativa, considerou-se o tempo de execução do modelo e a melhoria alcançada (*speedup*) no desempenho dos programas compilados. Espera-se que nossa abordagem seja capaz de encontrar sequências de otimização que alcancem um desempenho superior àquele obtido pelo melhor nível de otimização do compilador (-OX), baseando-se em apenas características estáticas de cada código.

1.1 Justificativa

Ashouri (ASHOURI et al., 2018) aponta que os compiladores são aperfeiçoados em apenas alguns pontos percentuais por ano. Isso mostra a importância de se estudar formas de aprimorar seu funcionamento. A otimização de código automática é um dos desafios ainda a ser resolvido e pode representar uma melhora significativa na qualidade do código compilado (COOPER et al., 2006). Com o grande crescimento das áreas de internet das coisas, entre outras linhas de computação, surge a demanda da utilização de sistemas embarcados e de arquiteturas específicas para cada situação. Essa variedade de arquiteturas dificulta o trabalho do programador em otimizar seu código para cada uma delas. Neste cenário, faz-se necessário um compilador que consiga definir, de forma automática, boas sequências de otimização. Considerando que esse processo de otimização costuma ocorrer uma única vez no projeto de sistemas embarcados (antes da entrega do produto final) e impacta significativamente em seu desempenho operacional, é razoável utilizar um método de busca que gaste tempo para encontrar melhores formas de se otimizar o código.

Assim, o desenvolvimento e aprimoramento de algoritmos para a seleção e ordenação de sequências de otimização é cada vez mais importante e influencia na qualidade de qualquer trabalho em computação que demanda otimização específica de um código em linguagem de alto nível para a linguagem de máquina da arquitetura alvo. Como o espaço de busca das sequências de passos de otimizações é muito grande, o problema é classificado como np-completo, e faz-se necessária a utilização de abordagens inteligentes e adaptativas para guiar a busca.

1.2 Objetivos

O objetivo do trabalho é desenvolver uma estratégia híbrida de exploração que seja mais rápida que abordagens iterativas e que encontre melhores sequências que as abordagens preditivas. Para isso, é preciso construir um conjunto de referência representativo, a partir do qual seja possível encontrar códigos similares aos programas alvo e cujas sequências sejam relevantes para melhorar o desempenho desses programas. Também é proposto avaliar diferentes estratégias para selecionar programas similares ao programa alvo e empregar um método capaz de definir automaticamente a quantidade de grupos (k) que devem ser retornados por um método de agrupamento, com o objetivo de aumentar a eficiência média na seleção destes programas similares. Também é necessário implementar uma abordagem que trate o problema de ordenação de passos de otimização como um problema de busca do menor caminho em um grafo, similar ao problema do caixeiro viajante, tornando-o mais adequado ao uso de algoritmos baseados em colônia de formigas.

1.3 Organização do texto

Além dessa introdução, o texto deste trabalho foi estruturado nos seguintes capítulos:

- Fundamentação teórica (Capítulo 2): Introduce conceitos básicos sobre compiladores, o problema de otimização de código, técnicas de redução do espaço de busca, otimização por algoritmos baseados em colônia de formigas, e mostra alguns trabalhos relacionados ao problema.
- Modelo Proposto (Capítulo 3): Explica o funcionamento do modelo proposto
- Experimentos (Capítulo 4): Descreve o ambiente de experimentação e apresenta os resultados obtidos
- Conclusão (Capítulo 5): Analisa os resultados dos experimentos e propõem trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo são introduzidos conceitos básicos de um compilador, de otimização de código, do problema de seleção e ordenação de passos de otimização, das formas de avaliar uma sequência, de métodos de agrupamento, e do algoritmo de colônia de formigas.

2.1 Compiladores e otimização

Um compilador tem como função principal transformar um código de uma linguagem para outra sem alterar a semântica do código, ou seja, para toda entrada, os dois programas devem produzir as mesmas saídas. (AHO; SETHI; LAM, 2008).

Um código produzido pelo compilador pode ser transformado para que gaste menos tempo de execução ou consuma menos espaço em memória. Isto é realizado através de análises e transformações no código chamadas de passos de otimização. "Otimização" é uma denominação exagerada, visto que as aplicações destas transformações não garantem como resultado o código com melhor desempenho (código ótimo), mas ainda assim, conseguem um aperfeiçoamento substancial em relação à versão original (AHO; SETHI; LAM, 2008) (MUCHNICK, 1997).

Cada compilador tem uma lista de passos de otimização possíveis de serem usadas. Entre elas, existem os passos de análise, que retiram informação do código, e os passos que aplicam transformações. São exemplos comuns de transformações: eliminação de sub expressões comuns, propagação de cópias, eliminação de código morto e transposição para constantes (AHO; SETHI; LAM, 2008).

Esses passos de otimização não necessariamente melhoram qualquer código, podendo até mesmo piorar o executável em alguns casos (JUNIOR; SILVA, 2015). Para minimizar este problema, é comum que compiladores ofereçam níveis de otimização, os quais são sequências fixas de passos formados a partir de um subconjunto de todos os passos disponíveis e que são aplicados em uma ordem pré-estabelecida definida pelo projetista do compilador. Tais sequências podem ser escolhidas usando comandos como -O, -O1, -O2 e -O3 (JUNIOR; SILVA, 2015). Mesmo que a escolha de quais passos utilizar e em qual ordem executa-las seja bem feita, e funcione para a maioria dos programas, não pode-se garantir que sempre serão as melhores sequências possíveis para todos os códigos. Portanto, buscar sequências de otimização específicas para o código e para a arquitetura alvo é uma boa forma de se conseguir melhor desempenho com o código compilado.

2.2 Problema de Seleção e Ordenação de Otimizações

Durante o processo de compilação, mais especificamente depois de gerado o código intermediário, o compilador pode aplicar uma série de operações (passos de otimização) que visam sua otimização. Por exemplo, o uso de alguns passos de otimização pode modificar o código de modo que ele faça menos leituras na memória, e, consequentemente, execute mais rapidamente, mas sem comprometer a sua semântica. Os diversos passos existentes podem ou não melhorar a eficiência de um código, dependendo de sua estrutura e comportamento. Assim, para que se consiga um código final melhor, é necessário escolher quais passos se utilizar para cada programa em questão.

Percebeu-se também, que há uma correlação entre esses passos, de tal forma que execução de um deles interfere na eficiência dos outros que serão aplicados posteriormente (JOHNSON, 2004). Johnson mostrou esse comportamento por meio de um exemplo utilizando duas transformações, alocação mínima de registradores e escalonamento de Instruções, como segue:

Considere o seguinte trecho de código, formado por duas operações de atribuição:

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

Ao serem convertidas para uma linguagem de baixo nível (por exemplo, *Assembly*), essas atribuições são traduzidas em uma operação de escrita e uma operação de leitura. Sem qualquer transformação, este código poderia ser convertido em:

```
ld t1, b  
st t1, a  
ld t2, c  
st t2, d
```

Sendo *ld* uma operação de leitura, e *st* uma operação de escrita. Ao aplicar a alocação mínima de registradores, o registrador utilizado na primeira operação de atribuição (*t1*) será utilizado novamente na segunda, gerando o seguinte código:

```
ld t1,b  
st t1,a  
ld t1,c  
st t1,d
```

E então, ao aplicar o escalonamento de instruções, nada será feito, pois a semântica do programa depende da ordem em que as instruções são executadas. Porém, se aplicarmos as operações de otimização na ordem inversa, teremos outro resultado. Pri-

meiramente, o escalonamento de instruções trocará a ordem das operações, de modo que as leituras fiquem juntas, seguidas pelas operações de escrita. Isso possibilita que elas sejam paralelizadas. Entretanto, isso impossibilita a troca dos registradores pelo alocação mínima. O código resultante é apresentado a seguir:

ld t1, b

ld t2, c

st t1, a

st t2, d

O paralelismo diminuirá o tempo de resposta do código, mostrando que aplicar a sequência de otimização nesta ordem é melhor em termos de velocidade. Porém, foram usados dois registradores diferentes, enquanto a primeira sequência de otimização produz um código que usa apenas um. Ou seja, ela é melhor em usar os registradores disponíveis de forma eficiente. Usar muitos registradores desnecessariamente pode diminuir a qualidade do código, pois se em um outro trecho do código for necessário guardar uma informação em um registrador e todos estiverem sendo usados, será necessário incluir instruções extras para guardar o conteúdo de algum registrador na memória para liberar o seu uso, além disso, quando o seu conteúdo for requisitado, outra instrução será necessária para escrevê-lo de volta em um registrador. Essas instruções adicionais, além de aumentar o tamanho do código, aumentam o tempo de execução, (MARTINS, 2016). Portanto, a escolha de qual sequência será melhor para o código é influenciada pela arquitetura alvo, tornando esse processo ainda mais difícil.

2.2.1 Decidibilidade do problema

Em (TOUATI; BARTHOU, 2006) foi demonstrado que achar uma sequência s tal que quando aplicada em um programa p gere um outro, semanticamente similar, mas com desempenho ótimo é indecidível. Essa demonstração é feita reduzindo o já conhecido e indecidível problema da linguagem vazia.

Problema da linguagem vazia (SIPSER, 2007): Considerando L uma linguagem aceita por uma Máquina de Turing M , é possível determinar se o conjunto de palavras aceitas por esta linguagem é vazio?

Primeiro, Touati e Barthou (2006) reformulam o problema de encontrar a sequência como:

Problema de ordenação de passos modificada: Dado qualquer programa, uma forma de avaliá-lo, um conjunto de dados de entrada, e um limiar de desempenho para a métrica avaliada (*threshold*), existe ao menos uma sequência de otimização que ao aplicá-la no programa, resulte em uma nova versão cuja avaliação seja menor ou igual ao

limite proposto?

Ao assumir que existe ao menos um programa p que, com infinitas sequências de otimização diferentes, pode-se transformá-lo em infinitos p' diferentes, em (TOUATI; BARTHOU, 2006) são comparadas sequências de otimização de um conjunto de passos disponíveis, com sequências de letras de um alfabeto. Com base nisso, os autores reduzem o problema da linguagem vazia no problema de ordenação de passos. A prova completa pode ser encontrada em (TOUATI; BARTHOU, 2006).

Os autores propõem diferentes formas de simplificação do problema a fim torná-lo decidível. Para isso, é preciso limitar o número de sequências no espaço de busca, e adicionar uma função de custo da compilação e um limite para esse custo. Essa função pode ser, por exemplo, o número máximo de passos em uma sequência, o tempo de compilação quando usando a sequência, o número de passos distintos em uma sequência ou o número de sequências diferentes exploradas.

2.2.2 O Espaço de Busca

Ser decidível não garante que o problema possa ser resolvido em um tempo razoável. O problema de ordenação de passos, mesmo quando limitado por uma função de custo, ainda é um problema intratável, ou seja, para certos tamanhos de entrada, pode não haver uma forma de resolver o problema em um tempo aceitável com as tecnologias disponíveis atualmente.

Considerando apenas a seleção de quais dos passos de otimização serão utilizados, sem analisar a ordem em que serão executados, pode-se definir o tamanho do espaço de busca em função da quantidade q de passos distintos como 2^q . Para o problema de ordenação de passos de otimização, o tamanho do espaço de busca é da ordem de $n!$. Ambos crescem por uma função não polinomial, tornando-os problemas intratáveis. Quando os dois problemas são analisados em conjunto, ou seja, achar a melhor sequência ordenada de tamanho n , sendo possível escolher passos dentre um conjunto de tamanho q , teremos um espaço de busca de q^n .

Este tamanho inviabiliza uma busca exaustiva pela melhor sequência de passos de otimização. Uma busca manual, feita pelo desenvolvedor do código, é demorada e depende de sua experiência com a tarefa. Assim, faz-se necessárias abordagens inteligentes e automáticas para direcionar a busca, usando uma heurística, e métodos que reduzam este espaço de busca.

2.2.3 Formas de redução do espaço de busca

Neste trabalho foram empregadas duas técnicas, KNN e k-médias, a fim de selecionar códigos similares e utilizar suas sequências para simplificar o espaço de busca,

reduzindo o problema em uma tarefa de ordenação apenas.

O KNN (FACELI et al., 2011) é um algoritmo de classificação, normalmente usado para indicar em qual dos grupos previamente definidos um elemento pertence. Dado um parâmetro K , são selecionados os K elementos mais próximos do alvo, seguindo o critério de distância euclidiana. A classificação é dada pelo grupo com maior frequência dentre estes k elementos.

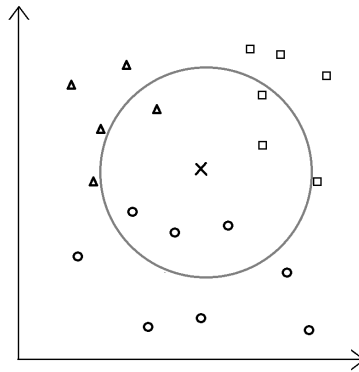


Figura 1 – Funcionamento do algoritmo KNN

No exemplo ilustrado na Figura 1, há 3 grupos, representados pelos círculos, quadrados e triângulos. O símbolo X representa o elemento alvo, e o algoritmo foi executado para $k = 6$. Como o grupo maior representado dentre os 6 mais próximos é o dos círculos, o algoritmo classifica o alvo como um círculo.

Diferente de um algoritmo de classificação, um algoritmo de agrupamento (*clustering*) tem o objetivo de agrupar uma base de dados não classificada, baseando-se nas características de cada objeto, de tal forma que membros de um mesmo grupo sejam semelhantes. Por não ser necessário fornecer as classes ao algoritmo, ele é considerado um algoritmo de aprendizado não-supervisionado (JAIN; DUBES et al., 1988).

Para se calcular o quão semelhantes são dois objetos, o algoritmo de agrupamento deve ter uma função de distância, que, baseado no vetor de características de dois objetos, determine um valor de similaridade (ou dissimilaridade) entre os dois. Uma forma comum e simples de se calcular esse valor é usar a distância euclidiana entre os vetores de atributos dos objetos, em que cada característica é tratada como uma dimensão no espaço (JAIN; DUBES et al., 1988). A distância euclidiana entre dois objetos A e B , com N características, em que uma característica i de A é representada por A_i , e uma

característica i de B é representada por B_i , é calculada por:

$$Distância(A,B) = \sqrt{\sum_{i=0}^N (A_i - B_i)^2} \quad (2.1)$$

K-Means (JAIN; DUBES et al., 1988) ou K-Médias é um algoritmo de agrupamento em que o número de grupos K é dado como parâmetro. O algoritmo tenta encontrar K grupos de forma a minimizar a soma das distâncias euclidianas entre cada objeto e o ponto médio de seu grupo

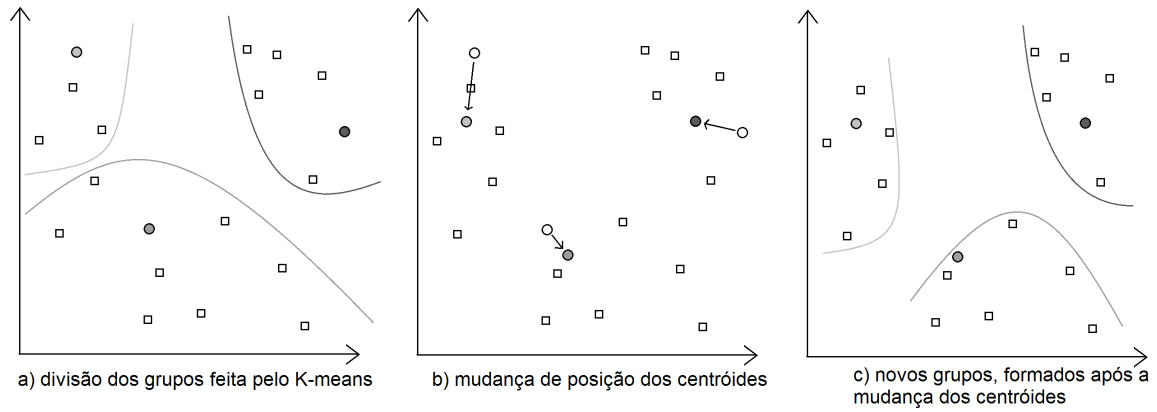


Figura 2 – Funcionamento do algoritmo K-Means

No início do algoritmo, são escolhidos K pontos aleatórios, cada um representando um grupo, e os objetos são alocados ao grupo do ponto mais próximo. O algoritmo funciona de forma iterativa, como ilustrado na Figura 2. Os círculos coloridos representam os pontos médios de cada grupo, e as linhas delimitam quais objetos (representados pelos quadrados) pertencem aos grupos, como ilustrado na Figura 2 (a). Ao início de uma iteração, é calculado o centroide C de cada grupo, o qual tem cada uma de suas dimensões definidas por:

$$C_i = \frac{\sum_{j=0}^N E_i^j}{N} \quad (2.2)$$

Em que N é o número de elementos no grupo, e E_i^j é o valor da dimensão i do elemento de número j do grupo.

Esses pontos calculados serão os novos pontos médios de cada grupo. No gráfico da Figura 2 (b), é mostrada a diferença das posições dos centroides de duas iterações anteriores (em branco) e atual (em tons de cinza). Os grupos são redefinidos, como representado na Figura 2 (c), associando cada objeto ao grupo do ponto médio (centroide) mais próximo. O algoritmo repete esse processo (iteração) até que nenhum objeto mude de grupo. (JAIN; DUBES et al., 1988)

2.2.4 Otimização por Colônia de Formigas

O algoritmo de colônia de formigas (DORIGO; MANIEZZO; COLORNI, 1996) pode ser usado para resolver diferentes problemas de otimização combinatorial. Seu funcionamento é inspirado no comportamento de uma colônia de formigas reais que, ao achar uma fonte de comida, conseguem estabelecer o caminho mais curto entre ela e o formigueiro.

Ao percorrer o caminho do formigueiro à fonte de alimento, uma formiga deixa um rastro de feromônio por onde passa. Assim, durante a procura por comida, quando a formiga se depara com mais de uma alternativa de caminho, ela usa a quantidade de feromônio depositada em cada via para ajudá-la na decisão de qual deles seguir. Nesse caso, o caminho com mais feromônio tem maior chance de ser escolhido. Esse método propicia a convergência para o caminho mais curto com o tempo, pois as formigas que pegam esse caminho terminam seu trajeto mais rápido, deixando seu rastro de feromônio e atraindo mais formigas para aquele caminho. O feromônio evapora com o tempo, fazendo com que os caminhos com menos formigas fiquem com cada vez menos feromônio, até que somente o caminho ótimo possua uma quantidade significativa de feromônio (DORIGO; MANIEZZO; COLORNI, 1996). A Figura 3 ilustra esse processo de exploração das formigas.

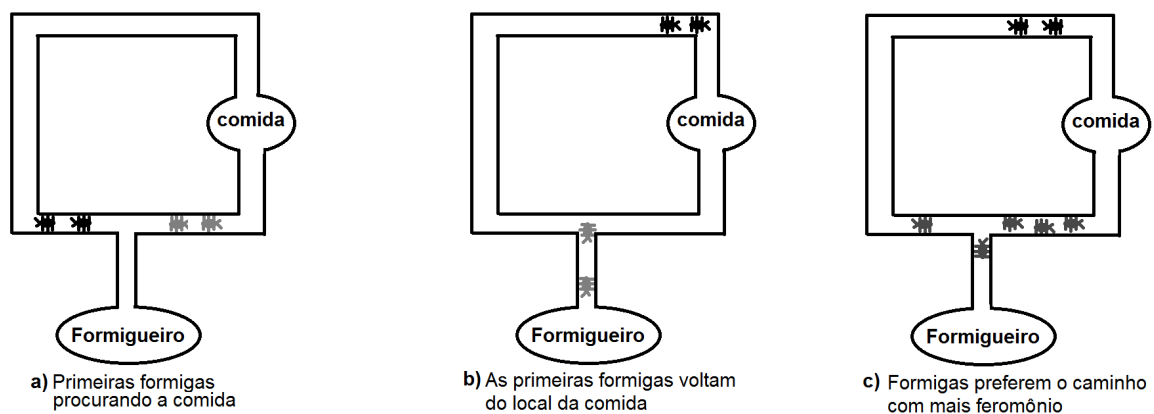


Figura 3 – Comportamento de formigas em procura alimento

Na Figura 3 (a), as formigas começam na parte escolhendo aleatoriamente o caminho pelo qual percorrer. Como ilustrado na Figura 3 (b), as formigas que percorreram o caminho mais curto (da direita) voltaram ao formigueiro em menos tempo, deixando seu rastro de feromônio. Assim, as próximas formigas que saem do formigueiro a procura de comida, escolhem com maior frequência o caminho da direita, como pode ser visto na Figura 3 (c), pois ele possui uma maior quantidade de feromônio.

O comportamento do algoritmo tem algumas diferenças do método das formigas

reais. O mais notável é que ele funciona com tempo discreto. No algoritmo, o caminho entre o formigueiro e a comida leva sempre o mesmo tempo, um ciclo de execução, independente do tamanho do caminho escolhido (solução construída). Apesar disso, a quantidade de feromônio que a formiga deixará pelo caminho é inversamente proporcional a distância percorrida ($\frac{1}{d_{ij}}$). As formigas do algoritmo percorrem um grafo, e há uma tabela de feromônios com uma entrada para cada aresta (parte usada na construção de uma solução), onde é guardado sua quantidade de feromônio. (DORIGO; MANIEZZO; COLONI, 1996).

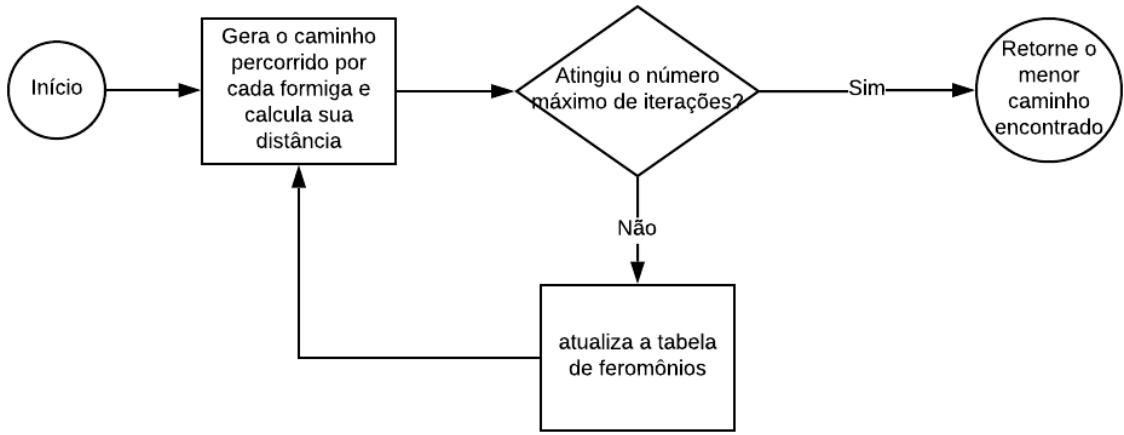


Figura 4 – Funcionamento básico de um algoritmo baseado em colônia de formigas

Para gerar o caminho (construção de uma solução) de uma formiga k na iteração t , o algoritmo calcula, em cada etapa do caminho (tomada de decisão), uma probabilidade $p_{i,j}^k(t)$ para cada possível parte a ser adotada (cada aresta a_{ij} que pode ser percorrida pela formiga), sendo i o vértice atual da formiga e j o vértice destino. Então, é escolhida a aresta com base nas respectivas probabilidades. A probabilidade da formiga k escolher o caminho de i para j no instante t ($p_{ij}^k(t)$) é definida como:

$$(2.3) \quad p_{i,j}^k(t) = \begin{cases} \frac{[\tau_{i,j}(t)]^\alpha * [\eta_{i,j}]^\beta}{\sum_{k \in K} [\tau_{i,k}(t)]^\alpha * [\eta_{i,k}]^\beta}, & \text{se } j \in K \\ 0, & \text{caso contrário} \end{cases}$$

Sendo K o conjunto das arestas permitidas no momento para a formiga k , $\eta_{i,j}$ a visibilidade, calculada por $1/d_{i,j}$; $\eta_{i,j}$ o feromônio entre i e j ; e $\alpha e \beta$ parâmetros do ACO que controlam a importância do feromônio e da distância entre os vértices na escolha do caminho, respectivamente.

A atualização da quantidade de feromônios é feita em duas partes: evaporação e deposição de feromônio, e é definida por:

$$\tau_{i,j} = \rho \tau_{i,j} + \Delta \tau_{i,j} \quad (2.4)$$

Em que $\tau_{i,j}$ é a quantidade de feromônio na aresta $a_{i,j}$, ρ é um parâmetro (taxa de evaporação) do algoritmo e $\Delta \tau_{i,j}$ é definido por:

$$\Delta \tau_{i,j} = \sum_{k=1}^m \Delta \tau_{i,j}^k \quad (2.5)$$

Onde m é a quantidade de formigas e $\Delta \tau_{i,j}^k$ é a quantidade de feromônio depositada pela formiga k na aresta $a_{i,j}$, dada por:

$$\Delta \tau_{i,j}^k = \begin{cases} \frac{Q}{d_k}, & \text{se a formiga } k \text{ usa a aresta } a_{i,j} \\ 0, & \text{caso contrário} \end{cases} \quad (2.6)$$

Sendo Q um parâmetro do ACO e d_k a distância total percorrida pela formiga.

2.3 Trabalhos Relacionados

Nessa sessão são mostrados alguns trabalhos encontrados na literatura que tratam do problema de seleção e ordenação de otimizações.

Em (NOBRE, 2017) foi desenvolvido uma forma de se construir um grafo direcional, em que os vértices são passos usados por sequências boas conhecidas e os vértices ligando arestas i e j recebem pesos de acordo com a frequência que i aparece logo antes de j nas sequências. um sistema modular foi feito em LARA para controlar e guiar compilação com diferentes algoritmos de busca iterativa. Os testes são feitos majoritariamente em *benchmarks* e processadores alvos relacionados a sistemas embarcados. Foi desenvolvido um algoritmo baseado em Recozimento simulado, que tem como vantagem o baixo gasto de memória, por não precisar guardar informação de sequências já exploradas. O nosso trabalho se diferencia de (NOBRE, 2017) por usar um método iterativo diferente, o algoritmo baseado em colônia de formigas.

Em (JUNIOR, 2016) é apresentada uma abordagem híbrida, com uma fase preditiva, e uma iterativa. Primeiramente, o algoritmo utiliza uma SVM para selecionar códigos similares em uma base de referência, e então, cria uma população inicial de sequências de otimização com as sequências destes códigos. Esta população é usada como a população inicial de um algoritmo genético. A abordagem conseguiu *speedup* médio superior, porém não significativo, a uma busca puramente iterativa (2.115 contra 2.074).

Em (ASHOURI et al., 2016) os autores optaram por uma abordagem preditiva e utiliza-se de um modelo de rede *bayesiana* para fazer a predição das sequências de otimização. Experimentos foram feitos no compilador GCC, e compilados para uma plataforma embarcada ARM. Os resultados são comparados com vários modelos de aprendizado de máquina do estado da arte. no artigo são avaliadas três formas diferentes de se caracterizar os programas: através de características estáticas, dinâmicas e ambas ao mesmo tempo. Os melhores resultados foram obtidos usando as características dinâmicas, seguido das características híbridas. Foram utilizados os *benchmarks* *Polybench* e *cBench*, e segundo os autores, a abordagem consegue *speedups* em média de 1.2x em relação a um método aleatório de busca de sequências, 1.37x em relação a um método iterativo baseado em aprendizado de máquina e 1.48x em relação a modelos preditivos.

Em (MARTINS et al., 2016) é utilizado um método de mineração de dados a uma representação simbólica (baseada em DNA) do código para selecionar códigos similares ao código alvo e diminuir o espaço de busca. Diferentes abordagens de exploração iterativa foram avaliadas e os experimentos mostraram que o método consegue melhorar significativamente o desempenho dos códigos dos *benchmark* utilizados (Texas Instruments) em relação a versão não otimizada, alcançando um *speedup* médio próximo a abordagem iterativa (Algoritmo genético tradicional), mas com uma melhora significativa em relação ao tempo de exploração. Assim como nesse trabalho, os experimentos foram feitos usando um processador para sistemas embarcados, o LEON3.

Em (AGAKOV et al., 2006) foram utilizados dois modelos de aprendizado de máquina para reduzir o espaço de busca, um modelo de distribuição identicamente distribuídas (*independent identically distributed - IID*) e um modelo de Markov. Os modelos foram avaliados usando os compiladores TI C6713 e AMD Au1500 e o *benchmark* usado para os experimentos foram 12 códigos do UTDSP. Segundo os autores, o modelo conseguiu um *speedup* médio de 1.22x para o primeiro compilador e 1.27x para o segundo em relação aos códigos originais, avaliando apenas duas sequências para cada código. Esse trabalho difere do nosso pois não é feita uma reordenação das sequências encontradas pelos modelos preditivos.

Em (XAVIER; SILVA, 2018) é proposto um modelo híbrido que utiliza de um algoritmo de agrupamento (*farthest first*) para selecionar códigos similares ao código alvo, e então utiliza um algoritmo baseado em colônia de formigas (ACO) para reordenar estas

sequências. Para adaptar o problema de ordenar uma sequência para ser resolvido pelo ACO, é criado um grafo a partir dos passos encontrados nas sequências dos códigos similares, utilizando a ordem em que estes passos aparecem para definir os pesos das arestas. Este modelo é melhor explicado na seção 3. O nosso trabalho é baseado no modelo apresentado em (XAVIER; SILVA, 2018), diferenciando-se por usar características estáticas ao invés de dinâmicas para caracterizar os códigos, por avaliar diferentes técnicas de seleção de códigos similares (KNN e K-Means) e por propor uma forma diferente de avaliar as soluções construídas pelas formigas (através da compilação simulação da execução do código alvo utilizando a sequência), enquanto a avaliação em (XAVIER; SILVA, 2018) é definida pela distância percorrida no grafo.

3 Modelo Proposto

Neste trabalho, foi desenvolvido um método híbrido para a exploração de sequências de otimização específicas para o código em compilação (alvo). Isto é, uma técnica preditiva é usada para diminuir o espaço de busca, e então, uma técnica iterativa é usada para explorar este espaço reduzido. Dividimos o sistema em duas etapas: uma fase *off-line* de preparação da base de referência, a qual é formada pelas características estáticas e a melhor sequência encontrada para cada um dos códigos do *benchmark*; e uma fase operacional, na qual busca-se a melhor sequência para um código alvo. A Figura 5 representa o funcionamento geral do sistema. A seguir, cada um destes passos são descritos com mais detalhes.

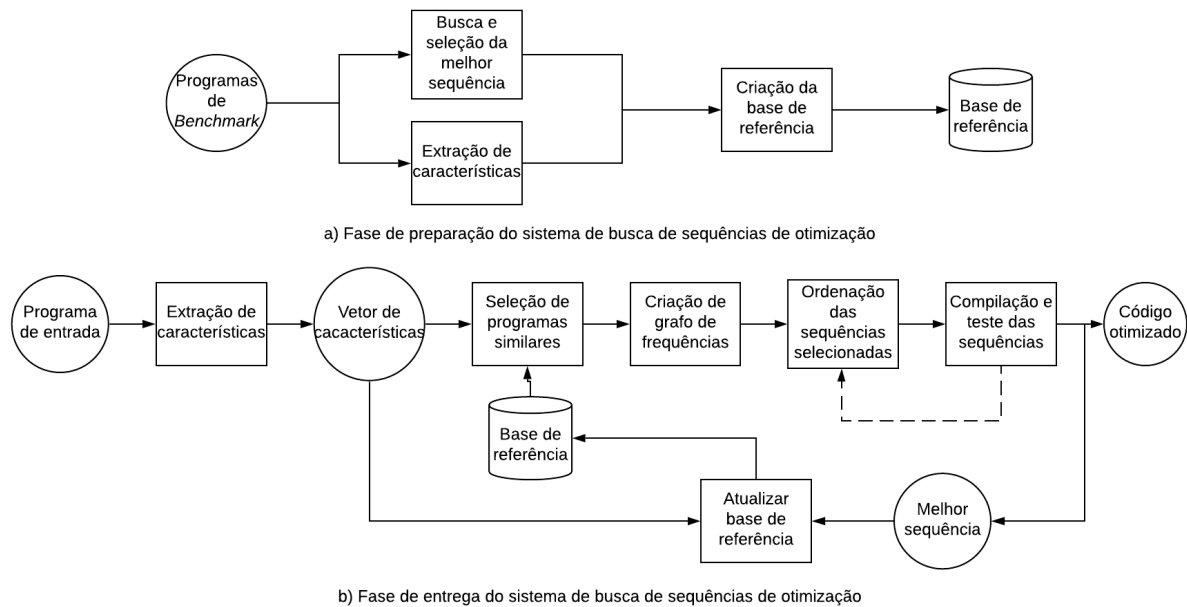


Figura 5 – Esquematização do funcionamento das fases de preparação e entrega do sistema de otimização automático de código

A fase de preparação (Figura 5 (a)) é simples e consiste na extração de um vetor de características estáticas de cada um dos programas do *benchmark* e aplicação de um método de busca iterativa por um extenso período de tempo a fim de encontrar uma sequência de passos mais próxima possível da ótima. Com os pares formados pelo vetor de características do código e sua melhor de otimização sequência, é criada a nossa base de dados de boas sequências, a qual será usada para diminuir o espaço de busca do método iterativo. É esperado que esta base, criada inicialmente, seja incrementada continuamente com novos pares de programas e sequências de passos de otimização com os novos códigos compilados.

Na fase operacional (Figura 5 (b)), o sistema recebe como entrada um novo código a ser compilado e retorna a sequência de passos de otimização que, dentre as exploradas, resultou no executável com o menor tempo de execução. Essa fase inicia com a extração do vetor de características estáticas do código alvo, o qual é usado para selecionar os k programas da base de referência mais similares. A partir das sequências desses programas similares, cria-se um grafo direcionado, chamado aqui de grafo de frequências, em que os vértices representam os passos de otimização presentes nas sequências e o peso de uma aresta entre os vértices i e j representa, proporcionalmente, a frequência com que i aparece antes do j nas sequências. Cada sequência dos programas similares é ordenada utilizando um algoritmo baseado em colônia de formigas, o qual é guiado pelo grafo de frequências. No final do processo, o código alvo é submetido às sequências dos programas similares e àquelas geradas a partir da ordenação. Os códigos resultantes (executáveis) são então avaliados (por meio de simulação) e a melhor destas é retornada como a saída do sistema, juntamente com o código compilado.

3.1 Extração de características

Diferente da forma feita em (XAVIER; SILVA, 2018), onde foram usadas características dinâmicas do código, nosso modelo utiliza características estáticas. Esta abordagem é mais simples já que não necessita da compilação do código para a extração das características. Apesar de as características estáticas não conterem toda a informação que se pode obter de forma dinâmica, esperamos que o sistema consiga selecionar programas similares apenas com elas. Para um trabalho futuro, é interessante realizar uma análise comparativa dessas abordagens (estática e dinâmica) a fim de identificar seu efeito na eficiência do método de exploração.

Como a nossa proposta é otimizar apenas a função mais custosa de um programa, retira-se as características apenas do código da função em questão. Isso é possível já que as características estáticas são retiradas do código intermediário, diferente das características dinâmicas, que são extraídas do código executável, o qual engloba todas funções do programa.

Para extrair as características, usamos um módulo extrator desenvolvido em (JUNIOR, 2016) e que foi gentilmente cedido pelos autores. A lista das características que podem ser obtidas através desse módulo é apresentada na Tabela 1

Tabela 1 – Características Estáticas, retiradas de (JUNIOR, 2016)

ID	Características estáticas
1	Número de instruções
2	Número de instruções de atribuição

Continuação da tabela 1	
ID	Características estáticas
3	Número de instruções binárias inteiras
4	Número de instruções binárias de ponto flutuante
5	Número de instruções de termino de bloco básico
6	Número de instruções binárias bit a bit
7	Número de instruções de vetor
8	Número de instruções de acesso e endereçamento de memória
9	Número de instruções agregadas
10	Número de instruções de conversão de inteiro
11	Número de instruções de conversão de ponto flutuante
12	Número de instruções de chamada
13	Número de instruções de chamada com ponteiros nos argumentos
14	Número de instruções de chamada que tem mais de 4 argumentos
15	Número de instruções de chamada que retornam inteiros
16	Número de instruções de chamada que retornam ponto flutuante
17	Número de instruções de chamada que retornam ponteiros
18	Número de instruções <i>Switch</i>
19	Número de instruções de desvios indiretos
20	Número de instruções de desvios condicionais
21	Número de instruções de desvios incondicionais
22	Número de instruções de <i>load</i>
23	Número de instruções de <i>store</i>
24	Número de instruções de <i>GetElemPtr</i>
25	Número de outras instruções
26	Número de nós PHI
27	Número de blocos básicos sem nós PHI
28	Número de blocos básicos com até 3 nós PHI
29	Número de blocos básicos com mais de 3 nós PHI
30	Número de nós PHI por bloco básico
31	Número médio de instruções por bloco básico
32	Número de arestas no grafo de fluxo de controle
33	Número de arestas críticas no grafo de fluxo de controle
34	Número de blocos básicos
35	Número de blocos básicos com 1 sucessor
36	Número de blocos básicos com 2 sucessores
37	Número de blocos básicos com mais de 2 sucessores
38	Número de blocos básicos com 1 predecessor
39	Número de blocos básicos com 2 predecessores

Continuação da tabela 1	
ID	Características estáticas
41	Número de blocos básicos com mais de 2 predecessores
42	Número de blocos básicos com 1 predecessor e 1 sucessor
43	Número de blocos básicos com 1 predecessor e 2 sucessores
44	Número de blocos básicos com 2 predecessores e 1 sucessor
45	Número de blocos básicos com 2 predecessores e 2 sucessores
46	Número de blocos básicos com mais de 2 predecessores e sucessores
47	Número de blocos básicos com menos de 15 instruções
48	Número de blocos básicos com número de instruções entre 15 e 500
49	Número de blocos básicos com mais de 500 instruções

Após a extração das características de cada programa, é realizado uma normalização dos dados. Isto é, para cada característica X , define-se $X_{máx}$ como o maior valor de X dentre os programas disponíveis (base de referência e código alvo), e divide-se o valor X de cada programa por $X_{máx}$, atualizando todos os valores como segue:

$$X' = \frac{X}{X_{máx}} \quad (3.1)$$

Desta forma, todos os atributos têm seus valores normalizados entre 0 e 1. Esse procedimento é realizado para que quando calculada a distância euclidiana entre dois elementos, uma característica com valores muito grandes não tenha um impacto maior que as de menor valor.

3.2 Seleção de programas similares

As características retiradas dos programas da base de referência e do código a ser compilado (código alvo) são empregadas para determinar um grupo de programas similares. Espera-se que as sequências desses programas também sejam boas para o novo código. Neste trabalho foram avaliadas duas formas de se executar essa tarefa, por meio de um método de agrupamento (*K-means*) e um de classificação (KNN - *k nearest neighbor*).

O KNN consiste em selecionar os K programas mais próximos do código alvo. Esta proximidade é definida pela distância euclidiana entre o vetor de características dos programas.

Para utilizar o *K-means*, o código alvo é adicionado à base de referência, a qual é submetida ao algoritmo de agrupamento. A partir dos K grupos retornados pelo algoritmo, adota-se como similares aqueles classificados no mesmo grupo do código alvo.

Após experimentações preliminares, foi decidido utilizar o método KNN. Uma explicação mais detalhada dos experimentos e a análise empregada nesta decisão é apresentada na Seção 4.2.

3.3 Construção do grafo

Após selecionadas as sequências dos programas similares ao código a ser compilado (alvo), é realizada a ordenação dos passos existentes nessas sequências usando um algoritmo baseado em colônia de formigas (ACO). O ACO foi originalmente concebido para encontrar um melhor caminho em um grafo (DORIGO; MANIEZZO; COLORNI, 1996), então, para usá-lo, precisamos transformar o nosso problema em um problema de busca em grafo. Assim como proposto em (XAVIER; SILVA, 2018), construiu-se um dígrafo completo em que cada vértice representa um passo de otimização que esteja presente em pelo menos uma das sequências selecionadas. O peso das arestas de um vértice i para um vértice j é definido como o número de vezes em que o passo i precede um passo j (não necessariamente adjacentes) nas sequências usadas para construir o grafo. Um exemplo de como isso ocorre é ilustrado na Figura 6.

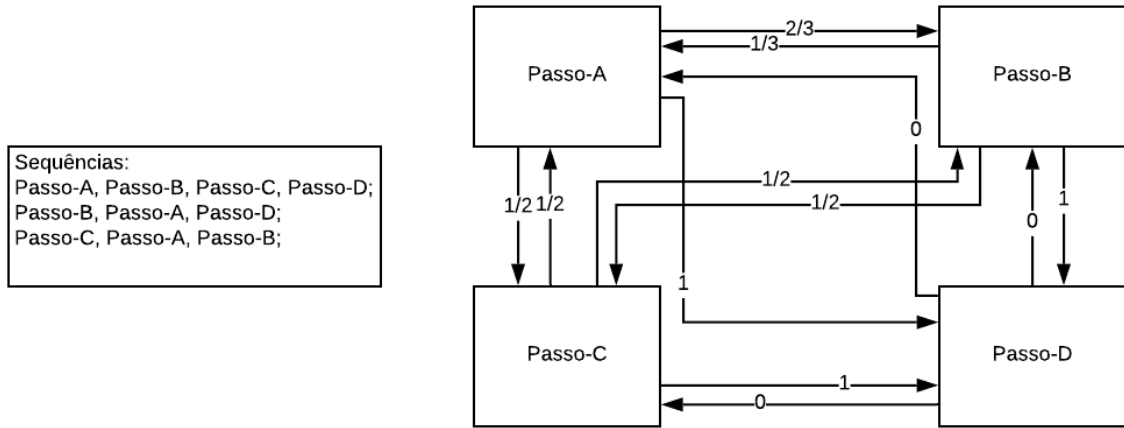


Figura 6 – Exemplo de grafo de frequência de passos de otimização

Nesse grafo, se o peso de uma aresta entre i e j for maior que o peso do vértice entre j e i , considera-se que aplicar o passo i antes de j é melhor, já que na maioria dos casos, i ocorre antes de j . O peso da aresta de i para j é definido por

$$peso(i,j) = \frac{freq(i,j)}{freq(j,i) + freq(i,j)} \quad (3.2)$$

Em que $freq(i,j)$ é a quantidade de sequências na qual i precede j . Na Figura 6, o peso da aresta entre o vértice do Passo-A e o vértice do Passo-B é de $2/3$, pois das 3

vezes em que esses passos aparecem na mesma sequência, em 2 o passo A precede o passo B. Desta forma, quanto maior o peso de uma aresta $a_{i,j}$, maiores são as chances de que A deva preceder B na sequência ótima. Esse grafo é usado como a heurística (visibilidade) do ACO. Assim, como o ACO foi originalmente concebido para achar o menor caminho, o custo de uma aresta $a_{i,j}$ com um peso alto, deve ter um custo baixo, então definimos $custo(i,j)$ como:

$$custo(i,j) = 1 - peso(i,j) \quad (3.3)$$

3.4 Ordenando as sequências selecionadas

Em nossa implementação, o algoritmo baseado em colônia de formigas deve ordenar a sequência de um dos programas similares de tal forma que a soma dos custos, calculados pelo grafo de frequências, seja o menor possível. O caminho de cada formiga é feito de tal forma que ela passe por cada passo presente na sequência de entrada exatamente uma vez. Assim, garantimos que o algoritmo não acrescenta nem retira nenhum passo das sequências, ou seja, ele só os re-ordena.

Após as soluções de todas as formigas serem avaliadas, é feita a atualização dos feromônios, como descrito na Seção 2.2.4 deste trabalho. A distância percorrida por uma formiga (d_k) é definida pela soma dos custos das arestas. Foram adotados para os parâmetros do ACO os valores usados em (XAVIER; SILVA, 2018). Eles são: $\alpha = 1$; $\beta = 5$; $\rho = 0.99$ e $Q = 100$.

Após as k sequências serem ordenadas, compilamos o código com as k sequências e avaliamos o desempenho dos executáveis resultantes. Como isso é feito antes e depois da fase de ordenação, o processo completo de exploração totaliza $2k$ compilações. A sequência que tiver melhor tempo de resposta é retornada como o resultado final do nosso sistema.

Durante experimentos preliminares, observou-se que as sequências que retornaram o menor custo nem sempre eram as que resultavam as menores latências. Um exemplo disso, é mostrado na Figura 7, na qual é mostrada a relação entre o custo de uma solução e a latência resultante ao compilar o código alvo.

Para contornar isso, uma segunda estratégia também foi avaliada, na qual a compilação e simulação do código com a sequência encontrada é usada para definir o custo da solução. Ou seja, adota-se a latência resultante como a métrica a minimizar. Para reduzir o número de vezes em que o código é compilado, foi adotada uma Tabela *hash* que, durante todo o funcionamento do algoritmo baseado em colônia de formigas, guarda o tempo de resposta de cada sequência avaliada. Desta forma, evita-se que seja necessário compilar mais de uma vez uma mesma sequência. Ainda assim, esse método tem um custo

Latência

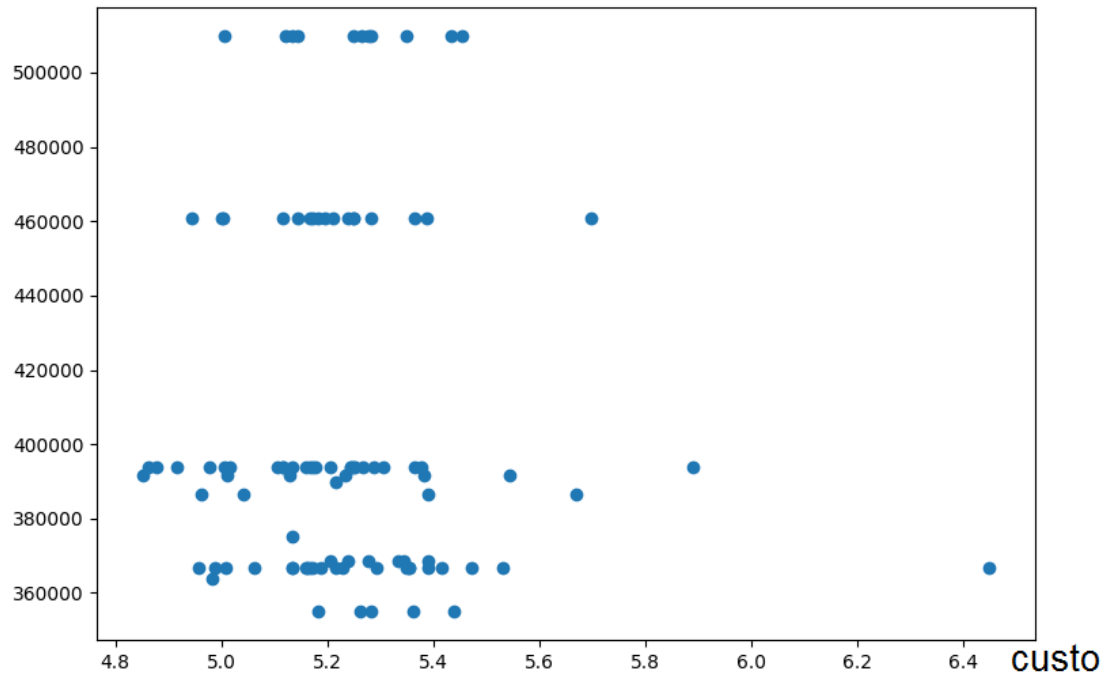


Figura 7 – Relação entre o custo e a latência de soluções avaliadas para o código *treesort* do *Benchmark* do LLVM

computacional bem mais alto que da estratégia original (calcular o custo pelo grafo de frequências), visto que a compilação do código é a tarefa mais demorada do processo e é necessário fazê-la uma vez para cada formiga que retorne uma solução que ainda não foi avaliada.

4 Experimentos

Neste capítulo, é descrito o ambiente utilizado na experimentação, as estratégias e métricas para avaliação dos resultados, as configurações e parâmetros usados nos algoritmos e uma avaliação dos resultados.

4.1 Ambiente de Experimentação

Como *Benchmark*, foram utilizados 51 dos programas escritos em linguagem C disponíveis no *Benchmark* do *Test-Suite* do LLVM versão 3.7.1 (LLVM, 2019). Os programas foram escolhidos com base no conjunto de 61 códigos selecionados em (PURINI; JAIN, 2013). Não foi possível utilizar todos os códigos do conjunto devido ao tempo limitado para a conclusão deste trabalho, o que impossibilitou a adequação dos 10 códigos restantes para o nosso ambiente de compilação e simulação. A Tabela 2 lista os programas utilizados em nossos experimentos.

Os programas foram compilados pelo LLVM (LLVM, 2019) para um processador para sistemas embarcados chamado LEON3, o qual adota uma arquitetura SPARC v.8, e o código de máquina foi avaliado por meio do simulador TSIM2 (GAISLER, 2019b), criado especificamente para esta arquitetura. Para automatizar a compilação e a simulação, foi

Tabela 2 – Programas do Benchmark

ackermann	flops8	partialsums
ary3	fpconvert	pi
bubblesort	heapsort	puzzle
chomp	himenobmtxpa	puzzlestanford
dry	huffbench	queens
fannkuch	intmm	queensmcgill
ffbench	lists	quicksort
fib2	lowercase	random
fldry	lpbench	realm
flops	mandel	reedsolomon
flops1	mandel2	richardsbenchmark
flops2	matrix	salsa20
flops3	methcall	simplehash
flops4	misr	spectralnorm
flops5	nbody	strcat2
flops6	objinst	towers
flops7	oscar	treesort

utilizado um ambiente (*framework*) baseado na linguagem LARA (CARDOSO et al., 2012). Esse ambiente é totalmente modular, de tal forma que é simples adicionar um novo algoritmo de exploração de sequências de otimização. Além disso, como a linguagem LARA é baseada em aspectos, as decisões de projeto são implementadas separadamente do código alvo e os componentes do processo de compilação são acionados através de comandos de alto nível, permitindo a abstração do compilador adotado (caixa preta).

Para avaliar nossa proposta, utilizamos a técnica *leave-one-out*, em que para cada código do benchmark a ser avaliado (compilado), considera-se todos os demais códigos como integrantes do conjunto de referência. Essa estratégia foi adotada por causa da baixa quantidade de códigos disponíveis (adaptados para o ambiente). Assim, o *leave-one-out* possibilitou a avaliação de um número satisfatório de códigos. Para avaliar o desempenho resultante da aplicação de cada sequência encontrada, o programa alvo também foi compilado e simulado sem qualquer otimização (-O0), bem como com as sequências padrões dos níveis de otimização do compilador LLVM (-O1, -O2 e -O3). O desempenho do código compilado no nosso modelo (*speedup*) é sempre comparado com a sequência que obteve melhor tempo de resposta, a qual é denominada -OX. O *speedup* é calculado como sendo a razão entre a latência obtida pelo -OX(L_{OX}) e a latência resultante do nosso modelo (L_M), como segue:

$$speedup = \frac{L_{OX}}{L_M} \quad (4.1)$$

4.2 Análise das abordagens de seleção de programas similares

Por meio de experimentos preliminares realizados sobre o conjunto de programas utilizando o método de *K-means*, observou-se que alguns dos grupos formados pelo algoritmo ficaram muito maiores. Por exemplo, para $k = 6$, o algoritmo construiu um grupo com 29 programas, um grupo com 21, dois grupos com 3 e dois com apenas um programa. Para $k = 9$ há um comportamento similar, sendo formado um grupo com 16 programas, um grupo com 15 programas, um grupo com 13 programas, um grupo com 6 programas, um grupo com 3, um grupo com 2 e 3 grupos com 1 programa cada.

Além desse problema dos tamanhos dos grupos, pode-se perceber que, para a nossa proposta, é mais vantajoso que sejam selecionados os programas mais similares ao código alvo, o que acontece com o algoritmo *KNN*, mas não é garantido no *K-means*, que se preocupa em agrupar os programas em relação aos centroides. A Figura 8 exemplifica a diferença entre as técnicas.

Nos gráficos da Figura 8, o quadrado cinza representa o programa alvo, enquanto os outros quadrados indicam os programas da base de referência. Pode-se notar no exemplo apresentado que, mesmo que o método de agrupamento conseguisse formar grupos de

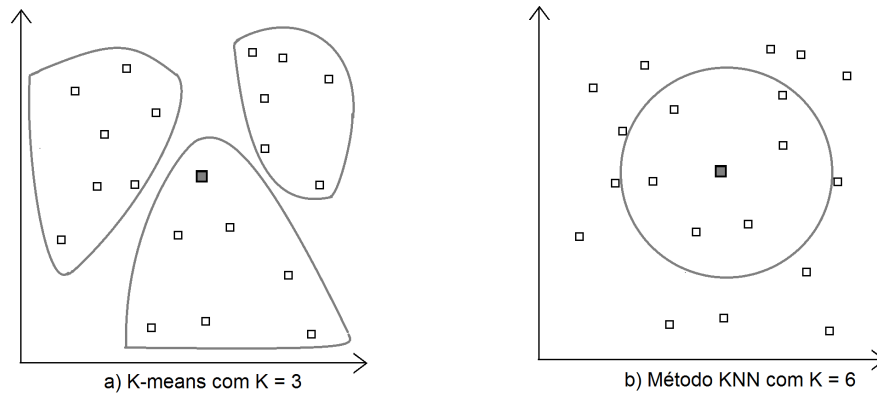


Figura 8 – Comparação entre K-Means e KNN para selecionar programas similares

tamanhos parecidos, não é garantido que os programas no mesmo grupo do código alvo sejam os mais similares a ele. Com o método KNN, a métrica de distância euclidiana (neste caso, entre os pares de vetores de características) assegura que os programas selecionados são os mais próximos (similares) dentre os programas disponíveis na base de referência.

Esses experimentos também permitiram observar se apenas usando o método de seleção de programas similares, é possível melhorar o código analisado. A Tabela 3 mostra os resultados encontrados nesses experimentos.

Tabela 3 – Resultado dos experimentos preliminares com o algoritmo KNN

K	<i>Speedup</i>			Desvio padrão	Execução (em segs)	Nro. Testes	Qtde. códigos
	Menor	Maior	Médio				
1	0.014	1.111	0.942	0.169	1.37	1	18
2	0.014	1.111	0.964	0.153	2.54	2	22
3	0.774	3.436	1.041	0.348	3.72	3	24
6	0.774	3.436	1.041	0.346	7.41	6	26
9	0.774	3.436	1.045	0.346	10.99	9	27
12	0.774	3.436	1.047	0.345	14.62	12	29
15	0.774	3.436	1.051	0.344	18.36	15	30

Nas Tabelas 3 e 4, apresentamos diferentes métricas referentes aos experimentos preliminares realizados com os algoritmos *KNN* e *K-Means*, respectivamente. A primeira coluna indica o valor utilizado para o parâmetro K de cada algoritmo. As três colunas seguintes indicam o menor, o maior e o *speedup* médio obtido dentre o *benchmark* utilizado. A quinta coluna indica o desvio padrão, a sexta indica o tempo de execução médio, em segundos, no computador utilizado para os experimentos. A coluna *Nro. Testes* mostra a quantidade de sequências testadas em cada execução. Este número é importante pois está diretamente relacionado ao tempo de execução, já que a compilação e avaliação de

Tabela 4 – Resultados dos experimentos preliminares com o algoritmo *K-Means*

<i>K</i>	<i>Speedup</i>			Desvio padrão	Execução (em segs)	Nro. Testes	Qtde. códigos
	Menor	Maior	Médio				
20	0.579	1.112	0.986	0.081	6.72	6	19
19	0.579	1.112	0.989	0.085	9.60	7	19
14	0.810	3.385	1.049	0.357	10.54	9	21
12	0.838	3.385	1.054	0.355	14.33	12	24
9	0.843	3.385	1.054	0.346	14.90	14	25
6	0.910	3.385	1.065	0.344	28.28	27	32
5	0.910	3.385	1.065	0.344	28.15	27	32

uma sequência é a tarefa com maior custo computacional do algoritmo. A coluna *Qtde. códigos* mostra a quantidade de códigos do *benchmark* para os quais o algoritmo conseguiu encontrar uma sequência que provê *speedup* maior que 1. Na Tabela 3 nota-se que para *K* acima de 3, o método *KNN* alcança um *speedup* médio acima de 1, ou seja, é capaz de melhorar o desempenho do código executável gerado em relação a melhor sequência usada pelos níveis de otimização do compilador. Pode-se observar também que o *speedup* médio não tem aumento expressivo com valores de *K* maiores que 3, apesar de aumentar o tempo de execução.

Na Tabela 4, a coluna *Nro. Testes* foi preenchida com o valor médio do número de testes para cada valor de *K* utilizado. Isso foi necessário já que diferentemente do método utilizando *KNN*, os grupos têm tamanhos diferentes entre si, logo o número de testes varia entre os códigos. É possível perceber que o método utilizando o *K-Means* alcança um desempenho médio acima de 1 apenas quando avaliadas 9.3 sequências, enquanto o *KNN* precisa de apenas 3 para conseguir este resultado. Também foi observado durante os experimentos que o algoritmo de *K-Means* gera alguns grupos contendo apenas um código. Isso compromete a viabilidade do método, já que para os códigos que pertencem a um desses grupos não há códigos similares, impossibilitando o funcionamento da nossa abordagem. A quantidade desses grupos de tamanho 1 aumentam quando aumenta-se a quantidade de grupos. Em nossos experimentos, 2 dos 51 códigos analisados pertenciam a um grupo de tamanho 1 quando usado $K = 6$, portanto, não foi possível executar o método para eles. Para grupos formados a partir de $K = 12$ e $K = 19$, esse tipo de situação ocorreu para um número maior de códigos (6 e 9 códigos, respectivamente).

Como a tarefa de maior custo computacional no processo de busca é a avaliação das sequências, os dois modelos de seleção testados foram comparados utilizando configurações em que a quantidade de sequências avaliadas são parecidas. Em nossos experimentos, o *speedup* médio utilizando o *K-Means* foi um pouco maior (1.045 com *KNN* e 1.049 com o *K-Means* utilizando 9 testes, por exemplo), porém, o algoritmo utilizando *KNN* consegue *speedup* maior que 1 em mais códigos (como visto na coluna *Qtde. códigos* das Tabelas 3 e 4). Em razão disso, e por não termos o tempo necessário para fazer os experimentos

com os dois algoritmos, resolvemos seguir com os experimentos utilizando o *KNN*.

4.3 Análise das estratégias de avaliação usadas no ACO

Para as duas estratégias propostas para a avaliação da solução de uma formiga, foram utilizados os mesmos valores para os parâmetros ρ (taxa de evaporação) = 0.99, $\alpha = 1$, $\beta = 5$ e $q = 100$. Estes valores são os mesmos usados em (XAVIER; SILVA, 2018). Para o algoritmo que utiliza o grafo de frequências, também foi adotado os mesmos valores para o número de ciclos e a quantidade de formigas, ou seja, 100 para os dois parâmetros (XAVIER; SILVA, 2018). Para o algoritmo que utiliza a compilação para avaliar as soluções das formigas, entretanto, foi necessário reduzir o valor desses parâmetros originais para 30, devido ao elevado custo da execução com os parâmetros originais (100 ciclos e 100 formigas), o que impossibilitaria sua utilização em nossos experimentos.

A fim de avaliar a eficácia e a eficiência dessas duas abordagens, seus resultados foram comparados com aqueles obtidos usando apenas o método de seleção de sequências próximas. O algoritmo que utiliza o grafo de frequências foi executado para três diferentes valores de k : 6, 9 e 12. Cada configuração do algoritmo foi executada 3 vezes, e os resultados estão apresentados na Tabela 5. Os *speedup's* de cada execução, para cada programa do *benchmark* podem ser encontrados nos Apêndices A, B e C.

Tabela 5 – Resultados das execuções do ACO com grafo de frequência

K	Menor Speedup	Maior Speedup	Média aritmética dos speedup's	Média Geométrica dos speedup's	Desvio Padrão	Tempo Médio Gasto	Quantidade De Sequências Testadas
6	0.823	3.407	1.050	1.026	0.341	166.089 s	12
9	0.824	3.402	1.054	1.030	0.339	247.277 s	18
12	0.873	3.385	1.057	1.034	0.337	427.241 s	24

Na Tabela 5, a coluna K se refere a quantidade de programas próximos selecionados pelo algoritmo de seleção. As colunas *Menor Speedup* e *Maior Speedup* mostram o menor e maior *Speedup* encontrado dentre os programas de *benchmark* utilizados. A *Média aritmética dos speedup's* e a *Média Geométrica dos speedup's* são respectivamente as médias aritmética e geométrica do *speedup* alcançado para todos os programas do *benchmark* e o *Desvio Padrão* é o desvio padrão dessa métrica. O *Tempo Médio Gasto* é a média do tempo que o algoritmo leva para executar para um programa. A *Quantidade De Sequências Testas* é a quantidade de sequências de otimização que foram avaliadas utilizando a compilação do código alvo, esse número é sempre o dobro do K pois nosso algoritmo sempre avalia cada uma das sequências obtidas pelo seletor de programas próximos e então avalia as sequências ordenadas (após a aplicação do ACO).

Como o tempo de execução do método que utiliza avaliação por compilação é muito grande, foram feitos os experimentos apenas com o valor de $k = 6$. Este valor de k foi escolhido baseado nos resultados do método que utiliza o gráfico de frequência para fazer a avaliação. Infelizmente, pelo alto tempo necessário para executar o ACO que utiliza simulação como avaliação, e pelo curto prazo de tempo para a conclusão deste trabalho, os modelos foram executados apenas três vezes. A média aritmética dos resultados de cada estratégia de avaliação empregada no algoritmo ACO, juntamente com os resultados obtidos quando utilizados apenas os métodos de seleção de códigos similares estão apresentados na Tabela 6. Os *Speedup*'s de cada execução para cada programa do *benchmark* podem ser encontrados no Apêndice D.

Tabela 6 – Resultados das execuções do ACO com ambos os tipos de avaliação e do KNN

Modelo	Menor <i>Speedup</i>	Maior <i>Speedup</i>	Média aritmét.	Média geomét.	Desvio padrão	Tempo de exec.	Qtd. de sequenc. avaliadas	Número de códig. otimizados
KNN	0.774	3.436	1.041	1.015	0.346	7.41s	6	28
ACO com grafo frequência	0.823	3.407	1.050	1.026	0.341	166.09s	12	25
ACO com simulação das seq.	0.825	3.573	1.073	1.046	0.365	3436.92s	5406	38.3

Pode-se observar na Tabela 6 que a variação dos valores da média aritmética do *speedup* obtido utilizando o mesmo valores de k é muito pequena (em torno de 0.009) entre o modelo que utiliza apenas o KNN e o modelo que utiliza o ACO com avaliação por grafo de frequência. Quando comparados o modelo que utiliza apenas o KNN e o modelo que utiliza o ACO com avaliação por simulação das soluções, a diferença nas médias aritméticas é maior (em torno de 0.032). A coluna Número de programas otimizados mostra o número de programas em que o algoritmo conseguiu uma sequência melhor que o -OX. O modelo que utiliza ACO com simulação das sequências consegue otimizar um número significativamente maior (38% a mais) que o método com grafo de frequência. A partir da Tabela 6, pode-se definir o melhor método a partir de dois critérios diferentes: Em relação a qualidade das sequências obtidas, o modelo com simulação das soluções é claramente melhor, principalmente pela quantidade de códigos otimizados (39 dos 51 totais). De outra forma, se analisarmos o tempo de execução, o método que utiliza apenas a seleção por KNN se mostra mais eficiente. O aumento de aproximadamente 22 vezes no tempo de exploração quando utilizado a ordenação pelo ACO com grafo de frequência é muito grande, enquanto o número de códigos otimizados apenas aumenta em 3 e o *Speedup* médio aumenta em 0.009.

5 Conclusão

Neste trabalho foi desenvolvida uma abordagem para a exploração de sequências de otimização específicas para cada código a ser compilado (código alvo). Essa abordagem é composta por dois módulos principais: um seletor e um ordenador.

O módulo seletor visa identificar, dentre um conjunto de códigos previamente selecionados (base de conhecimento), aqueles que são mais similares ao novo código que está sendo analisado. Para isso, avaliou-se o uso de duas técnicas de aprendizado de máquina, uma de classificação (KNN) e uma de agrupamento (K-Means). Através dos resultados experimentais, foi possível observar que o uso das sequências de programas similares ao código alvo provê melhora significativa no desempenho desse código, mesmo quando avaliado um número pequeno ($k = 3$) de sequências de otimização, ou seja, a mesma quantidade dos níveis de otimização do compilador LLVM, os quais deveriam ser testados para determinar o melhor deles (-OX). O uso de apenas características estáticas simplifica o processo de otimização, se comparado com abordagens que utilizam características dinâmicas, já que não é necessário executar o código para obter as características. Além disso, também foi possível concluir que o método KNN é o mais adequado para o nosso propósito, uma vez que ele consegue obter programas similares de forma mais efetiva que o método de agrupamento (K-Means). Isso ocorre pois o método KNN garante que os códigos retornados são os mais próximos do código alvo, enquanto o K-Means cria grupos em relação as centróides dos códigos.

O módulo ordenador tem como objetivo encontrar a melhor ordem de aplicação dos passos existentes em cada uma das sequências selecionadas no módulo seletor. A ideia é que, por ser similares, os programas devem usar os mesmos passos de otimização, mas as especificidades de cada código pode demandar uma alteração na ordem de aplicação. Em nosso modelo, adotamos um algoritmo baseado em colônia de formigas (ACO) para encontrar a melhor sequência para cada código. A fim de adaptar o problema ao ACO, o mesmo foi transformado em um problema de percorrer todos os vértices de um grafo (caixeiro viajante). Duas estratégias de avaliação de soluções foram implementadas. A primeira define a avaliação em função da distância percorrida pela solução no grafo. A segunda compila e simula a execução do código com a solução, e utiliza o desempenho (*speedup*) do código como sua forma de avaliação. Os experimentos realizados apontam que ordenar as sequências usando um grafo de frequências criado a partir das sequências dos programas similares não melhora significativamente o desempenho médio alcançado apenas com as sequências similares, escolhidas pelo módulo seletor. Porém, quando utilizado o resultado da compilação do código como forma de avaliar as soluções constituídas por cada formiga do algoritmo, é possível obter *speedups* mais significativos (1.073 contra

1.05), embora demande um tempo de exploração consideravelmente maior (60.1 minutos ao invés de 2.7).

Para trabalhos futuros, propõe-se fazer uma análise da parametrização do algoritmo baseado em colônia de formigas, procurando configurações que consigam convergir para boas sequências usando menos ciclos e formigas. Também é interessante avaliar o desempenho da nossa abordagem utilizando características dinâmicas, e/ou conjuntos híbridos de características dinâmicas e estáticas. Há também a possibilidade de estudar a relação entre cada característica do código e as sequências que o otimizam de forma eficiente, a fim de determinar quais destas características realmente contribuem para determinar quais sequências são boas para o código. No algoritmo que seleciona programas próximos, pode-se avaliar o uso de diferentes métricas de distância diferentes da distância euclidiana, que foi utilizada neste trabalho. No algoritmo de formigas utilizando a compilação do código para avaliação da sequência, é possível que utilizando uma técnica de *surrrogate*, o tempo de execução diminua consideravelmente.

Referências

- AGAKOV, F. et al. Using machine learning to focus iterative optimization. In: *Proceedings of the international symposium on code generation and optimization*. [S.l.: s.n.], 2006. Citado na página 22.
- AHO, A.; SETHI, R.; LAM, S. *Compiladores: princípios, técnicas e ferramentas*. [S.l.]: LONGMAN DO BRASIL, 2008. Citado 2 vezes nas páginas 8 e 13.
- ASHOURI, A. H. et al. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, ACM, 2018. Citado 3 vezes nas páginas 8, 9 e 11.
- ASHOURI, A. H. et al. A bayesian network approach for compiler auto-tuning for embedded processors. In: IEEE. *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. [S.l.], 2014. Citado na página 9.
- ASHOURI, A. H. et al. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016. Citado na página 22.
- BROWN, S. D. et al. *Field-programmable gate arrays*. [S.l.]: Springer Science & Business Media, 2012. v. 180. Citado na página 10.
- CARDOSO, J. M. et al. Lara: an aspect-oriented programming language for embedded systems. In: ACM. *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. [S.l.], 2012. p. 179–190. Citado na página 32.
- COOPER, K. D. et al. Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, Springer, 2006. Citado 2 vezes nas páginas 8 e 11.
- DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, man, and cybernetics, Part B: Cybernetics*, 1996. Citado 3 vezes nas páginas 19, 20 e 28.
- FACELI, K. et al. Inteligência artificial: Uma abordagem de aprendizado de máquina. 2011. Citado na página 17.
- GAISLER, C. *LEON3 simulator*. 2019. Acessado em 26/10/2019. Disponível em: <<https://www.gaisler.com/index.php/products/processors/leon3>>. Citado na página 10.
- GAISLER, C. *TSIM2 ERC32/LEON simulator*. 2019. Acessado em 09/07/2019. Disponível em: <<https://www.gaisler.com/index.php/products/simulators/tsim>>. Citado na página 31.
- JAIN, A. K.; DUBES, R. C. et al. *Algorithms for clustering data*. [S.l.]: Prentice hall Englewood Cliffs, 1988. v. 6. Citado 2 vezes nas páginas 17 e 18.
- JOHNSON, N. E. *Code size optimization for embedded processors*. Tese (Doutorado), 2004. Citado na página 14.

- JUNIOR, N. L. Q. Abordagem hibrida para mitigacao do pso. 2016. Citado 4 vezes nas páginas 5, 9, 22 e 25.
- JUNIOR, N. L. Q.; SILVA, A. F. da. The use of different strategies of search space reduction in mitigation of optimization selection problem. In: IEEE. *2015 Latin American Computing Conference (CLEI)*. [S.l.], 2015. Citado na página 13.
- LLVM. *The LLVM Compiler Infrastructure Project*. 2019. Acessado em 01/07/2019. Disponível em: <<https://llvm.org/>>. Citado 2 vezes nas páginas 10 e 31.
- MARTINS, L. G. A. *Exploração de sequências de otimização do compilador baseada em técnicas híbridas de mineração de dados complexos*. Tese (Doutorado) — Universidade de São Paulo, 2016. Citado 2 vezes nas páginas 9 e 15.
- MARTINS, L. G. A. et al. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, 2016. Citado 2 vezes nas páginas 8 e 22.
- MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. [S.l.]: Morgan Kaufmann Publishers Inc., 1997. Citado na página 13.
- NOBRE, R. J. F. Efficient target and application specific selection and ordering of compiler passes. 2017. Citado na página 21.
- PURINI, S.; JAIN, L. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013. Citado na página 31.
- SIPSER, M. *Introducao a teoria da computacao*. 2. ed. [S.l.]: Thomson Learning, 2007. Citado na página 15.
- TOUATI, S. A. A.; BARTHOU, D. On the decidability of phase ordering problem in optimizing compilation. In: *Proceedings of the 3rd conference on Computing frontiers*. [S.l.: s.n.], 2006. Citado 2 vezes nas páginas 15 e 16.
- XAVIER, T. C. de S.; SILVA, A. F. da. Exploration of compiler optimization sequences using a hybrid approach. *Computing and Informatics*, 2018. Citado 7 vezes nas páginas 10, 22, 23, 25, 28, 29 e 35.

Apêndices

APÊNDICE A – Resultados do modelo usando knn ($k = 6$) e avaliação por grafo de frequências

	<i>Speedup</i> com $k = 6$			
Programas	Exec. 1	Exec. 2	Exec. 3	Média
ackermann	1.0312	1.0312	1.0312	1.0312
ary3	1.0011	1.0011	1.0011	1.0011
bubblesort	0.9603	0.9603	0.9603	0.9603
chomp	0.9821	0.9762	0.9883	0.9822
dry	0.9984	0.9984	0.9984	0.9984
fannkuch	1.1246	1.1246	1.1246	1.1246
ffbench	1.0504	1.0058	1.0493	1.0351
fib2	0.9036	0.9036	0.9036	0.9036
fldry	1.0265	1.0265	1.0265	1.0265
flops	1.0183	1.0183	1.0183	1.0183
flops1	1.0002	1.0002	1.0002	1.0002
flops2	1.1116	1.1116	1.1116	1.1116
flops3	0.9998	0.9998	0.9998	0.9998
flops4	1.0001	1.0001	1.0001	1.0001
flops5	1.0581	1.0581	1.0581	1.0581
flops6	1.0637	1.0637	1.0637	1.0637
flops7	1.0006	1.0006	1.0006	1.0006
flops8	1.0603	1.0603	1.0603	1.0603
fpconvert	1.0000	1.0000	1.0000	1.0000
heapsort	1.0105	1.0105	1.0105	1.0105
himenobmtxpa	1.0010	0.9965	0.9965	0.9980
huffbench	0.9952	0.9952	0.9952	0.9952
intmm	1.0060	1.0060	1.0060	1.0060
lists	1.0891	1.0891	1.0891	1.0891
lowercase	0.9760	0.9760	0.9834	0.9785
lpbench	0.9915	0.9915	0.9915	0.9915
mandel	1.0006	1.0000	1.0006	1.0004
mandel2	1.0316	1.0316	1.0316	1.0316

matrix	1.0042	0.9990	0.9996	1.0009
methcall	1.0002	1.0002	1.0002	1.0002
misr	1.0382	1.0346	1.0288	1.0339
nbody	0.9873	0.9873	0.9873	0.9873
objinst	3.3850	3.4522	3.3850	3.4074
oscar	1.0131	1.0131	1.0131	1.0131
partialsums	1.0063	1.0063	1.0063	1.0063
pi	0.9967	0.9967	0.9967	0.9967
puzzle	0.9998	1.0000	0.9998	0.9999
puzzlestanford	0.8233	0.8233	0.8233	0.8233
queens	0.9467	1.0014	1.1352	1.0278
queensmcgill	0.9492	0.9492	0.9492	0.9492
quicksort	1.1078	1.1078	1.1078	1.1078
random	1.0000	1.0000	1.0000	1.0000
realmm	1.0364	1.0364	1.0364	1.0364
reedsolomon	0.9997	0.9997	0.9999	0.9997
richardsbenchmark	0.9910	0.9968	0.9968	0.9949
salsa20	0.9032	0.9032	0.8877	0.8980
simplehash	1.0142	1.0142	1.0142	1.0142
spectralnorm	0.9271	0.9271	0.9272	0.9271
strcat2	1.0005	1.0005	1.0005	1.0005
towers	0.9983	0.9591	0.9591	0.9722
treesort	0.9925	0.8437	0.8437	0.8933

APÊNDICE B – Resultados do modelo usando knn ($k = 9$) e avaliação por grafo de frequências

	<i>Speedup</i> com $k = 9$			
Programas	Exec. 1	Exec. 2	Exec. 3	Média
ackermann	1.0312	1.0312	1.0312	1.0312
ary3	1.0011	1.0012	1.0011	1.0012
bubblesort	0.9660	0.9660	0.9987	0.9769
chomp	0.9706	0.9935	0.9706	0.9782
dry	0.9984	0.9984	0.9984	0.9984
fannkuch	1.1246	1.1246	1.1246	1.1246
ffbench	1.0480	1.0004	1.0029	1.0171
fib2	1.0222	1.0222	1.0222	1.0222
fldry	1.0265	1.0265	1.0265	1.0265
flops	1.0155	1.0155	1.0155	1.0155
flops1	1.0002	1.0002	1.0002	1.0002
flops2	1.1116	1.1116	1.1116	1.1116
flops3	0.9998	0.9998	0.9998	0.9998
flops4	1.0001	1.0001	1.0001	1.0001
flops5	1.0581	1.0581	1.0581	1.0581
flops6	1.0637	1.0637	1.0637	1.0637
flops7	1.0006	1.0006	1.0006	1.0006
flops8	1.0602	1.0602	1.0602	1.0602
fpconvert	1.0000	1.0000	1.0000	1.0000
heapsort	1.0055	1.0000	1.0000	1.0018
himenobmtxpa	0.9965	0.9965	0.9965	0.9965
huffbench	1.0150	0.9957	0.9961	1.0023
intmm	1.0060	1.0060	1.0060	1.0060
lists	1.0891	1.0891	1.0891	1.0891
lowercase	0.9760	0.9760	0.9760	0.9760
lpbench	0.9940	0.9915	0.9915	0.9923
mandel	1.0000	1.0000	1.0000	1.0000
mandel2	1.0316	1.0316	1.0316	1.0316

matrix	1.0048	1.0048	0.9995	1.0030
methcall	1.0002	1.0002	1.0002	1.0002
misr	1.0293	1.0605	1.0605	1.0501
nbody	1.0079	1.0079	1.0079	1.0079
objinst	3.3850	3.4361	3.3850	3.4020
oscar	1.0137	1.0137	1.0137	1.0137
partialsums	1.0063	1.0063	1.0063	1.0063
pi	0.9967	0.9967	0.9967	0.9967
puzzle	1.0000	1.0000	1.0000	1.0000
puzzlestanford	0.8239	0.8239	0.8239	0.8239
queens	1.0359	0.9952	1.0014	1.0108
queensmcgill	0.9492	0.9492	0.9492	0.9492
quicksort	1.1129	1.1129	1.1129	1.1129
random	1.0000	1.0000	1.0000	1.0000
realmm	1.0365	1.0364	1.0364	1.0365
reedsolomon	1.0010	0.9997	1.0008	1.0005
richardsbenchmark	0.9958	0.9963	0.9983	0.9968
salsa20	0.8877	0.8877	0.8877	0.8877
simplehash	1.0142	1.0142	1.0142	1.0142
spectralnorm	0.9271	0.9271	0.9273	0.9272
strcat2	1.0005	1.0005	1.0005	1.0005
towers	0.9591	0.9591	0.9591	0.9591
treesort	0.9925	0.8437	1.0595	0.9652

APÊNDICE C – Resultados do modelo usando knn ($k = 12$) e avaliação por grafo de frequências

	<i>Speedup</i> com $k = 12$			
Programas	Exec. 1	Exec. 2	Exec. 3	Média
ackermann	1.0312	1.0312	1.0312	1.0312
ary3	1.0012	1.0011	1.0011	1.0012
bubblesort	0.9660	0.9660	0.9660	0.9660
chomp	0.9706	1.0075	0.9970	0.9917
dry	0.9984	0.9984	0.9984	0.9984
fannkuch	1.1246	1.1246	1.1246	1.1246
ffbench	1.0076	1.0170	1.0072	1.0106
fib2	1.0222	1.0222	1.0222	1.0222
fldry	1.0265	1.0265	1.0265	1.0265
flops	1.0155	1.0155	1.0155	1.0155
flops1	1.0002	1.0002	1.0002	1.0002
flops2	1.1116	1.1116	1.1116	1.1116
flops3	0.9998	0.9998	0.9998	0.9998
flops4	1.0001	1.0001	1.0001	1.0001
flops5	1.0581	1.0581	1.0581	1.0581
flops6	1.0637	1.0637	1.0637	1.0637
flops7	1.0006	1.0006	1.0006	1.0006
flops8	1.0602	1.0602	1.0602	1.0602
fpconvert	1.0000	1.0000	1.0000	1.0000
heapsort	1.0055	1.0055	1.0086	1.0066
himenobmtxpa	0.9965	0.9965	0.9965	0.9965
huffbench	0.9952	1.0031	1.0089	1.0024
intmm	1.0060	1.0060	1.0060	1.0060
lists	1.0891	1.0891	1.0891	1.0891
lowercase	1.0000	1.0000	1.0015	1.0005
lpbench	0.9915	0.9915	0.9915	0.9915
mandel	1.0000	1.0000	1.0000	1.0000
mandel2	1.0316	1.0316	1.0316	1.0316

matrix	1.0048	1.0038	1.0051	1.0046
methcall	1.0002	1.0002	1.0002	1.0002
misr	1.0379	1.0605	1.0379	1.0455
nbody	1.0079	1.0079	1.0079	1.0079
objinst	3.3850	3.3850	3.3850	3.3850
oscar	1.0230	1.0230	1.0230	1.0230
partialsums	1.0063	1.0063	1.0063	1.0063
pi	0.9967	0.9967	1.0000	0.9978
puzzle	1.0001	1.0001	1.0001	1.0001
puzzlestanford	0.8928	0.8928	0.8928	0.8928
queens	0.0000	1.0014	0.9805	0.6607
queensmcgill	0.9492	0.9492	0.9492	0.9492
quicksort	1.1129	1.1129	1.1129	1.1129
random	1.0000	1.0000	1.0000	1.0000
realmm	1.0364	1.0365	1.0364	1.0365
reedsolomon	0.9997	1.0008	0.9997	1.0000
richardsbenchmark	0.9944	0.9941	0.9941	0.9942
salsa20	0.8877	0.8877	0.8877	0.8877
simplehash	1.0142	1.0142	1.0142	1.0142
spectralnorm	0.9999	0.9999	0.9999	0.9999
strcat2	1.0005	1.0005	1.0005	1.0005
towers	0.9853	0.9835	0.9847	0.9845
treesort	1.0366	0.9872	0.8437	0.9558

APÊNDICE D – Resultados do modelo usando knn($k = 6$) e avaliação por compilação do código

	<i>Speedup</i> com $k = 6$		
Programas	Exec. 1	Exec. 2	Média
ackermann	1.0311	0.9987	1.0149
ary3	1.0011	1.0827	1.0419
bubblesort	0.9987	0.9987	0.9987
chomp	1.0366	1.0250	1.0308
dry	0.9984	0.9984	0.9984
fannkuch	1.1400	1.1519	1.1460
ffbench	1.0539	1.0652	1.0596
fib2	1.0221	1.0221	1.0221
fldry	1.0264	1.0264	1.0264
flops	1.0291	1.0217	1.0254
flops1	1.0002	1.0002	1.0002
flops2	1.1953	1.1956	1.1955
flops3	0.9999	0.9999	0.9999
flops4	1.0001	1.0001	1.0001
flops5	1.0580	1.0580	1.0580
flops6	1.0637	1.0637	1.0637
flops7	1.0005	1.0005	1.0005
flops8	1.0613	1.0613	1.0613
fpconvert	0.9999	0.9999	0.9999
heapsort	1.0263	1.0199	1.0231
himenobmtxpa	1.0159	1.0101	1.0130
huffbench	1.0325	1.0307	1.0316
intmm	1.0060	1.0060	1.0060
lists	1.0907	1.0912	1.0909
lowercase	0.9834	1.0032	0.9933
lpbench	1.0061	1.0003	1.0032
mandel	1.0006	0.9743	0.9875
mandel2	1.0316	1.0316	1.0316

matrix	1.0053	1.0029	1.0041
methcall	1.0002	1.0002	1.0002
misr	1.0799	1.0799	1.0799
nbody	1.0119	0.9842	0.9981
objinst	3.5730	3.5730	3.5730
oscar	1.0357	1.0230	1.0294
partialsums	1.0120	1.0000	1.0060
pi	1.0000	1.0002	1.0001
puzzle	1.0002	0.8259	0.9130
puzzlestanford	0.8260	1.1399	0.9830
queens	1.1399	0.9602	1.0501
queensmcgill	0.9497	1.1128	1.0312
quicksort	1.1128	1	1.0564
random	1	1.0364	1.0182
realmm	1.0364	1.0019	1.0191
reedsolomon	1.0018	1.0069	1.0044
richardsbenchmark	1.0053	0.9205	0.9629
salsa20	0.9031	1.0257	0.9644
simplehash	1.0251	0.9939	1.0095
spectralnorm	0.9272	1.0005	0.9638
strcat2	1.0005	1.0000	1.0003
towers	0.9982	1.0952	1.0467
treesort	1.0952	1.0952	1.0952

matrix	0.9359	1.0031	1.0031	0.9990	0.9990	0.9990	0.9990
methcall	0.9996	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002
misr	1.0293	1.0288	1.0288	1.0288	1.0293	1.0293	1.0293
nbody	0.9873	0.9873	0.9873	0.9873	1.0079	1.0079	1.0079
objinst	0.0145	0.0145	3.4361	3.4361	3.4361	3.4361	3.4361
oscar	1.0108	1.0073	1.0073	1.0073	1.0075	1.0075	1.0075
partialsums	1.0125	1.0125	1.0116	1.0125	1.0116	1.0116	1.0125
pi	0.7316	0.9967	0.9967	0.9967	0.9967	0.9967	0.9967
puzzle	0.9998	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002
puzzlestanford	0.8108	0.8195	0.8195	0.8233	0.8251	0.8821	0.9999
queens	0.7818	0.8159	0.8159	0.9525	0.9805	0.9805	0.9805
queensmcgill	0.9273	0.9273	0.9492	0.9497	0.9492	0.9497	0.9648
quicksort	1.0516	1.0516	1.1078	1.0140	1.1129	1.1129	1.1129
random	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
realmm	0.9730	0.9734	0.9869	1.0365	0.9886	0.9886	0.9886
reedsolomon	0.9961	0.9978	1.0030	1.0030	1.0030	1.0030	1.0030
richardsbenchmark	0.9981	0.9981	0.9856	0.9820	0.9855	0.9855	0.9855
salsa20	0.6007	0.9032	0.9032	0.9032	0.9032	0.9032	1.0692
simplehash	1.0135	1.0135	1.0135	1.0142	1.0142	1.0142	1.0040
spectralnorm	0.9074	0.9074	0.9074	0.9271	0.9271	0.9329	0.9329
strcat2	0.6621	1.0002	1.0002	1.0002	1.0005	1.0005	1.0005
towers	0.8168	0.7806	0.9722	0.9722	0.9722	0.9722	0.9722
treesort	0.8437	0.8437	0.7741	0.7741	0.7741	0.7741	0.7741