ALEXANDRE DOS SANTOS MIGNON

ML4JIT - UM ARCABOUÇO PARA PESQUISA COM APRENDIZADO DE MÁQUINA EM COMPILADORES JIT

São Paulo
2017

ALEXANDRE DOS SANTOS MIGNON

ML4JIT - UM ARCABOUÇO PARA PESQUISA COM APRENDIZADO DE MÁQUINA EM COMPILADORES JIT

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Ciências.

São Paulo
2017

ALEXANDRE DOS SANTOS MIGNON

ML4JIT - UM ARCABOUÇO PARA PESQUISA COM APRENDIZADO DE MÁQUINA EM COMPILADORES JIT

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Ciências.

Área de Concentração:

Engenharia de Computação e Sistemas Digitais

Orientador:

Prof. Dr. Ricardo Luis de Azevedo da Rocha

São Paulo
2017

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Mignon, Alexandre dos Santos
ML4JIT - Um Arcabouço para Pesquisa com Aprendizado de Máquina em Compiladores JIT / A. S. Mignon -- versão corr. -- São Paulo, 2017.
101 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Aprendizado de Máquina 2.Compiladores JIT 3.Otimização de Código I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

RESUMO

Determinar o melhor conjunto de otimizações para serem aplicadas a um programa tem sido o foco de pesquisas em otimização de compilação por décadas. Em geral, o conjunto de otimizações é definido manualmente pelos desenvolvedores do compilador e aplicado a todos os programas. Técnicas de aprendizado de máquina supervisionado têm sido usadas para o desenvolvimento de heurísticas de otimização de código. Elas pretendem determinar o melhor conjunto de otimizações com o mínimo de interferência humana. Este trabalho apresenta o ML4JIT, um arcabouço para pesquisa com aprendizado de máquina em compiladores JIT para a linguagem Java. O arcabouço permite que sejam realizadas pesquisas para encontrar uma melhor sintonia das otimizações específica para cada método de um programa. Experimentos foram realizados para a validação do arcabouço com o objetivo de verificar se com seu uso houve uma redução no tempo de compilação dos métodos e também no tempo de execução do programa.

Palavras-Chave – otimização de código, compiladores JIT, aprendizado de máquina.

ABSTRACT

Determining the best set of optimizations to be applied in a program has been the focus of research on compile optimization for decades. In general, the set of optimization is manually defined by compiler developers and apply to all programs. Supervised machine learning techniques have been used for the development of code optimization heuristics. They intend to determine the best set of optimization with minimal human intervention. This work presents the ML4JIT, a framework for research with machine learning in JIT compilers for Java language. The framework allows research to be performed to better tune the optimizations specific to each method of a program. Experiments were performed for the validation of the framework with the objective of verifying if its use had a reduction in the compilation time of the methods and also in the execution time of the program.

Keywords – code optimization, JIT compilers, machine learning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AST *Abstract Syntax Tree* - Árvore Sintática Abstrata

IR *Intermediate Representation* - Representação Intermediária

JIT *Just In Time*

JVM *Java Virtual Machine* - Máquina Virtual Java

ML *Machine Learning* - Aprendizado de Máquina

RVM *Research Virtual Machine*

SVM *Support Vector Machine*

SUMÁRIO

1 INTRODUÇÃO

Compiladores e linguagens de alto nível são peças fundamentais para a construção de infraestruturas complexas e ubíquas de software. A crescente complexidade das máquinas e software, a introdução de processadores multi-núcleos e a preocupação com a segurança dos sistemas estão entre os problemas mais graves que devem ser tratados hoje pelos desenvolvedores de compiladores. Diante disso, um importante papel que a tecnologia de compiladores deve desempenhar, dentre outros, é a otimização de programas (HALL; PADUA; PINGALI, 2009).

Segundo Aho et al. (2006), otimização de código é a eliminação de instruções desnecessárias no código objeto, ou a substituição de uma sequência de instruções por uma sequência de instruções mais rápidas que tem o mesmo comportamento. A maioria dos compiladores comerciais possui otimizadores de código. Estes otimizadores geralmente proporcionam um bom nível de desempenho e, em alguns casos, o desempenho do código gerado está perto do pico de desempenho da máquina alvo. Entretanto, as configurações de otimizações de um compilador são definidas de forma manual, baseada em heurísticas adotadas pelos seus desenvolvedores. Alcançar bons resultados com sintonia manual, especialmente em códigos de larga escala, é uma tarefa difícil, cara e propensa a erros (HALL; PADUA; PINGALI, 2009). E, quando o compilador precisa ser utilizado em novas plataformas, as configurações de otimizações podem requerer ajustes ou uma completa redefinição. Além disso, manter as configurações de otimização é uma tarefa difícil porque alterações em uma configuração de otimização podem melhorar o desempenho de algumas aplicações enquanto degrada o de outras (LAU et al., 2006).

Para tratar tais questões, técnicas de aprendizado de máquina (*machine learning*) supervisionado têm sido usadas em compiladores para o desenvolvimento de heurísticas de otimização de código (STEPHENSON et al., 2003; CAVAZOS; MOSS, 2004; FURSIN et al., 2011). Cavazos e O'boyle (2006) aplicaram a técnica de *logistic regression* para determinar automaticamente quais otimizações são melhores para cada método de um programa desenvolvido em linguagem Java. Como avanço desse trabalho, Sanchez et

al. (2011) utilizaram a técnica de *Support Vector Machine* (SVM) como mecanismo de treinamento e aplicaram em uma máquina virtual de grande porte: a Testarossa da IBM.

Em geral, os trabalhos relacionados ao aprendizado de máquina em compiladores utilizam técnicas de aprendizado supervisionado (MITCHELL, 1997). Com o uso destas técnicas o aprendizado é feito através de exemplos fornecidos por algum supervisor externo em uma fase de treinamento. No caso do uso de aprendizado supervisionado em compiladores, o agente aprendiz é treinado para selecionar as melhores otimizações em relação à plataforma de execução e ao conjunto de programas usados para o treinamento.

Este trabalho apresenta o `ML4JIT`, um arcabouço para pesquisa com aprendizado de máquina em compiladores JIT para a linguagem Java. Com o uso desse arcabouço pretende-se descobrir o melhor conjunto de otimizações que deve ser aplicado a cada um dos métodos de um programa Java. Ele permite a realização de experimentos controlados e a utilização de diferentes algoritmos de aprendizado de máquina.

1.1 Estado da Arte

Esta seção apresenta uma breve descrição de trabalhos relacionados ao uso de aprendizado de máquina em compiladores. Eles serviram, de alguma forma, como base e motivação para a elaboração deste trabalho.

Heurísticas são usadas em compiladores otimizadores para guiar o processo de transformações do código para um determinado objetivo. Em geral, essas heurísticas são definidas manualmente pelos desenvolvedores do compilador. Este processo é caro, consome muito tempo e é propenso a erros, e pode levar a um desempenho sub-ótimo (HOSTE; GEORGES; EECKHOUT, 2010). Para lidar com tais questões, o aprendizado de máquina em compiladores foi aplicado inicialmente com o objetivo de automatizar o processo de geração de heurísticas (CAVAZOS; O'BOYLE, 2006; SANCHEZ et al., 2011).

Um dos primeiros trabalhos utilizando aprendizado de máquina em compiladores foi para lidar com transformações *looping unrolling* (MONSIFROT; BODIN; QUINIOU, 2002). Este trabalho utilizou o compilador GNU FORTRAN alterando-se a heurística de ativação de *loop unrolling* para utilizar um modelo aprendido através de exemplo. O algoritmo de aprendizado utilizado era baseado em árvores de decisão, o que permitia que o modelo gerado pudesse ser interpretado por um especialista.

McGovern, Moss e Barto (2002) apresentaram resultados usando *rollouts* e aprendizado por reforço para construir heurísticas para agendamento de blocos básicos. Na

simulação, o agendador usando aprendizado por reforço superou um agendador comercial em muitos *benchmarks* e teve um bom desempenho nos outros.

Stephenson et al. (2003) utilizaram programação genética para percorrer o espaço das possíveis funções de prioridade (associadas a cada transformação) para três transformações: hiper-blocos, alocação de registradores e leitura antecipada. Resultados mostraram pequenas melhorias nos *benchmarks* testados.

Cavazos e O'boyle (2006) aplicaram a técnica de *logistic regression* para habilitar ou desabilitar transformações em planos de compilação na Jikes RVM para a compilação específica de método. Estudos apresentados mostraram que as transformações poderiam obter melhores resultados se fossem selecionados planos de compilação específicos para cada método do programa. A Jikes RVM foi modificada para compilar métodos individuais variando as transformações aleatoriamente, uma medição de tempo foi adicionada a cada método e os planos de compilação com melhor tempo de execução foram selecionados e usados para o treinamento da *logistic regression*. A versão utilizando aprendizado de máquina superou a configuração padrão presente na Jikes RVM.

Hoste, Georges e Eeckhout (2010) apresentam uma proposta de sintonia automática de planos de compilação em um compilador JIT baseada em uma busca evolucionária com múltiplos objetivos. A ideia é aplicar uma sintonia fina no compilador para cenários específicos: uma dada plataforma de *hardware*, um conjunto de aplicações, ou um conjunto de entradas para aplicações de interesse. A sintonia inicia com uma exploração dos planos de compilação para descobrir quais deles são ótimo de Pareto (*Pareto-optimal*), e então um subconjunto deles é atribuído ao compilador JIT.

Fursin et al. (2011) apresentaram o MILEPOST GCC, um arcabouço de compilação iterativa que pode se auto ajustar em plataformas diferentes. As heurísticas no compilador são ajustadas para um conjunto de aplicações em uma plataforma alvo, dependendo do objetivo do usuário: melhorar o desempenho de execução ou o consumo de energia em plataformas embarcadas, por exemplo. O modelo de aprendizado de máquina é externo ao compilador e ele deve ser criado para cada aplicação.

Sanchez et al. (2011) aplicaram a técnica de *Support Vector Machine* (SVM) para habilitar ou desabilitar transformações em planos de compilação em uma máquina virtual de grande porte, a Testarossa da IBM, para a compilação específica de método. O processo de modificação dos planos de compilação partia do plano original da Testarossa realizando pequenas alterações ao longo do tempo em busca de melhorias locais. O desempenho de vazão não sofreu alterações, mas foi possível melhorar o desempenho de inicialização.

Kulkarni e Cavazos (2012) apresentaram uma técnica que automaticamente seleciona a melhor ordem prevista de transformações para métodos diferentes de um programa. A técnica utiliza uma rede neural artificial para predizer a ordem de transformações que é mais benéfica para um determinado método. As redes neurais artificiais são automaticamente inferidas usando *NeuroEvolution for Augmenting Topologies* (NEAT). A técnica foi implementada na máquina virtual Jikes RVM e apresentou melhorias de desempenho em um conjunto de *benchmarks* se comparado com a aplicação de transformações em uma ordem fixa.

Leather, Bonilla e O'boyle (2014) apresentaram um mecanismo para automaticamente buscar características de um programa que tem um maior impacto na qualidade de heurísticas de aprendizado de máquina. O espaço de características é descrito por uma gramática e é então percorrido com programação genética. Esse mecanismo foi aplicado para a transformação *loop unrolling* no compilador GCC.

Magni, Dubach e O'Boyle (2014) estudaram a transformação denominada *thread-coarsening* que junta duas ou mais *threads* paralelas aumentando a quantidade de trabalho realizada por uma única *thread*, reduzindo o total de *threads* instanciadas. Os autores utilizaram uma técnica baseada em redes neurais para selecionar automaticamente o melhor fator de *coarsening* e decidir também se é benéfico aplicar a transformação. Experimentos realizados utilizando esta técnica apresentaram um desempenho melhor do que usando a transformação de maneira tradicional.

1.2 Motivação

Conforme apresentado na seção anterior, existem diversos trabalhos que utilizam aprendizado de máquina em compiladores. Entretanto, a tentativa de reprodução de alguns trabalhos apontou diversos desafios e dificuldades. Em geral, os autores utilizam uma máquina virtual Java específica e a modificam para possibilitar a realização dos experimentos. Essas modificações nem sempre estão disponíveis ou mesmo detalhadas para permitir a reprodução do ambiente de pesquisa. Há casos ainda em que a JVM utilizada é proprietária e seu código fonte não é aberto, impossibilitando que ela seja utilizada facilmente por pesquisadores.

Outro ponto a ser considerado é que boa parte dos trabalhos apresentam resultados de experimentos realizados com um único tipo de algoritmos de aprendizado de máquina. Com isso, não se sabe ao certo se outros tipos de algoritmos seriam mais adequados ao

problema que se está tratando.

Por esses motivos, decidiu-se desenvolver um arcabouço de código aberto e livre para permitir a realização de pesquisas com otimização de código em compiladores JIT para a linguagem Java com o auxílio de aprendizado de máquina.

1.3 Objetivos

O objetivo principal deste trabalho é fomentar a realização de pesquisas para se descobrir uma melhor sintonia dos parâmetros de configuração de um compilador JIT. Essa sintonia visa permitir que diferentes planos de compilação sejam aplicados a cada um dos métodos do programa. Pretende-se com isso, reduzir o tempo de execução e/ou o tempo de compilação do programa.

Para auxiliar no processo de descoberta do melhor plano de compilação específico para cada método do programa, são utilizados algoritmos de aprendizado de máquina. Este trabalho possibilita que sejam analisados diferentes tipos de algoritmos permitindo assim a comparação dos resultados obtidos com cada um deles.

Compiladores JIT foram escolhidos como ferramenta de pesquisa, pois compilam o código em tempo de execução, permitindo coletar informações de perfil de execução e guiar o compilador para produzir um código melhor durante a execução do programa.

1.3.1 Objetivos Específicos

Como objetivos específicos tem-se:

- Realizar pesquisas com diferentes tipos de algoritmos de aprendizado de máquina para o objetivo principal proposto, na tentativa de se descobrir qual deles gera um melhor resultado para o problema em questão;
- Desenvolver um recurso para a instrumentação de código com o propósito de extrair informações de perfil de execução do programa, como o tempo de compilação e o tempo de execução de cada método do programa;
- Extrair informações que possam ser utilizadas para descrever os métodos do programa. Essas informações são usadas para auxiliar no treinamento e na predição dos algoritmos de aprendizado de máquina;

- Explorar diferentes conjuntos de otimizações para cada método de um programa, na tentativa de se descobrir qual deles gera um maior ganho no desempenho total do programa;

- Desenvolver uma estrutura para permitir a realização de pesquisas experimentais com aprendizado de máquina dinâmico. Com a aplicação desse tipo de aprendizado pretende-se reduzir o tempo gasto para o treinamento do modelo de aprendizado e também permitir que esse modelo se adapte mais rapidamente a mudanças na plataforma de execução do programa.

1.4 Contribuições

A principal contribuição deste trabalho é a estruturação de um laboratório para possibilitar que pesquisas experimentais sejam realizadas na tentativa de se descobrir o melhor conjunto de otimizações que deve ser aplicado de forma específica a cada método de um programa Java, com a intenção de reduzir o tempo de execução e/ou o tempo de compilação do programa. Para auxiliar neste processo de descoberta são utilizados algoritmos de aprendizado de máquina.

Como contribuições adicionais desse trabalho podem-se citar:

- Um recurso para a extração de características dos métodos de um programa Java. Essas características descrevem os métodos de um programa e são usadas pelos algoritmos de aprendizado de máquina;

- Um recurso para a coleta de dados de tempo de execução e tempo de compilação dos métodos de um programa Java, através da instrumentação do *bytecode* do método, em tempo de execução. Esse recurso foi desenvolvido para permitir que o processo de instrumentação do *bytecode* seja independente de uma máquina virtual Java específica;

- Um ambiente que permite que diferentes tipos de algoritmos de aprendizado de máquina sejam analisados, na tentativa de se descobrir qual deles gera um melhor resultado para o problema em questão;

- Uma proposta de estrutura para permitir a realização de pesquisas experimentais com aprendizado de máquina dinâmico na seleção do melhor conjunto de otimizações. Essa estrutura visa possibilitar que técnicas de aprendizado ativo ou aprendizado por reforço sejam analisadas e testadas com o arcabouço;

- Uma técnica, baseada em *nano-patterns*, que permite identificar automaticamente métodos de um programa que possam ser excluídos de um processo de instrumentação de código.
- Um arcabouço de código aberto e livre para a realização de pesquisas experimentais com aprendizado de máquina em compiladores JIT.

1.5 Organização

Este trabalho está organizado da seguinte forma: O capítulo 2 apresenta os conceitos básicos de compiladores, otimização de código e compilação JIT. O capítulo 3 apresenta os conceitos básicos sobre aprendizado de máquina, utilizados nesse trabalho.

O capítulo 4 apresenta uma visão geral do arcabouço `ML4JIT`, e como ele pode ser utilizado para pesquisas com aprendizado de máquina em compiladores. O capítulo 5 apresenta detalhes de projeto e implementação do arcabouço. O capítulo 6 apresenta os experimentos realizados para a validação do arcabouço e os seus resultados.

Por fim, o capítulo 7 apresenta as considerações finais do trabalho, as conclusões e os trabalhos futuros.

2 COMPILADORES

Este capítulo apresenta os conceitos relacionados a compiladores necessários para a compreensão deste trabalho. Inicialmente, apresenta-se uma visão geral sobre processadores de linguagem e a estrutura clássica de um compilador. A seguir, apresenta-se uma introdução sobre otimização de código e compilação JIT.

2.1 Visão Geral

Compiladores são programas de computador que transformam uma linguagem de entrada - a linguagem fonte - em uma linguagem de saída - a linguagem alvo. Geralmente, a linguagem fonte é um texto escrito em uma linguagem de programação que representa um programa e a linguagem alvo é uma linguagem de máquina compreendida diretamente pelo computador (AHO et al., 2006; COOPER; TORCZON, 2011).

Interpretadores são outro tipo comum de processadores de linguagem. Ao invés de produzir um programa alvo como resultado da tradução, um interpretador obtém como entrada uma especificação executável e produz como saída o resultado da execução da especificação.

O código de máquina produzido por um compilador é usualmente mais rápido que a execução do programa em um interpretador. Um interpretador, entretanto, pode usualmente fornecer melhores diagnósticos de erro que um compilador, porque ele executa o programa fonte instrução por instrução.

Processadores da linguagem Java combinam compilação e interpretação, como apresentado na Figura 1. Um programa fonte Java é primeiramente compilado em uma forma intermediária chamada de *bytecodes*. Os *bytecodes* são então interpretados por uma máquina virtual.

Uma máquina virtual pode ser considerada um mecanismo de execução de programas por *software*. Ao invés do programa executar diretamente no processador do computador

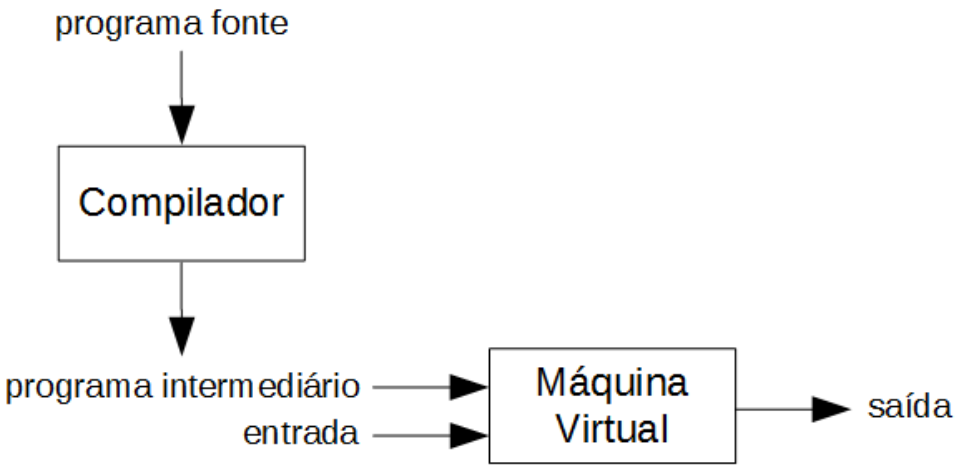

Figura 1: Um compilador híbrido. Adaptada de (AHO et al., 2006).

ele é executado por um *software*, a máquina virtual, que simula algum processador. A máquina virtual é responsável por interagir com o *hardware*. Um benefício da execução de um programa através de uma máquina virtual é permitir a portabilidade de programas entre diferentes plataformas. Por exemplo, *bytecodes* compilados em uma máquina podem ser interpretados em outra máquina.

Com o objetivo de se obter um melhor desempenho, alguns compiladores Java, denominados de compiladores JIT, traduzem os *bytecodes* em código de máquina nativo do *hardware* subjacente, imediatamente antes deles executarem o programa intermediário para processar a entrada. A seção 2.4 apresenta mais detalhes sobre compilação JIT.

2.2 Estrutura de um Compilador

A estrutura de um compilador é dividida basicamente em duas partes: *front-end* e *back-end*. O *front-end* lida com a linguagem fonte, enquanto o *back-end* lida com a linguagem alvo. A conexão entre as duas partes é feita por uma estrutura formal, denominada *representação intermediária* (*Intermediate Representation* - IR), que representa o programa em uma forma intermediária, independente da linguagem fonte e alvo. Para melhorar a tradução um compilador pode opcionalmente conter um *otimizador* que analisa e reescreve a IR (COOPER; TORCZON, 2011).

A Figura 2 apresenta a estrutura geral de um compilador composto dessas três partes. O *front-end* tem por objetivo entender o programa na linguagem fonte e codificá-lo na representação intermediária para depois ser usada pelo *back-end*. O *otimizador* obtém como entrada a IR do programa e produz outra IR semanticamente equivalente como

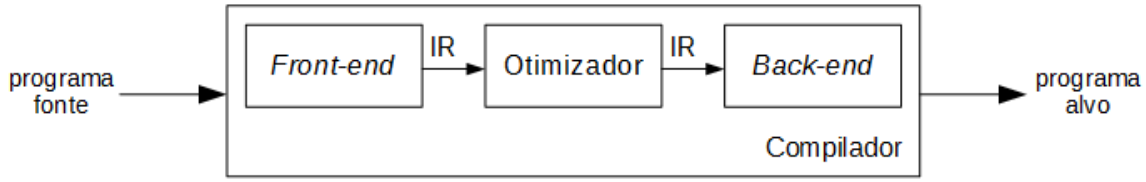

Figura 2: Estrutura geral de um compilador com três partes. Adaptada de (COOPER; TORCZON, 2011).

saída. Ele pode reescrever a IR para auxiliar o *back-end* a produzir um programa alvo mais rápido e/ou de menor tamanho de código. O *back-end* tem como objetivo mapear a IR em um conjunto de instruções da máquina alvo.

As partes da estrutura geral do compilador podem ser divididas em módulos para facilitar a compreensão e o desenvolvimento do compilador. A Figura 3 apresenta os módulos de um compilador. O *front-end* e o *back-end* [1] são compostos por cinco módulos cada um. Adicionalmente, o compilador contém módulos para manipulação da tabela de símbolos e relatório de erros; estes módulos são chamados por quase todos os outros módulos (GRUNE et al., 2012). A seguir, apresenta-se uma breve descrição de cada um dos módulos.

O *módulo de entrada do texto do programa* obtém o arquivo texto contendo o código fonte do programa, realiza a sua leitura, e o transforma em um fluxo (*stream*) de caracteres. Ele trabalha em cooperação com o sistema operacional e com o módulo de análise léxica.

O *módulo de análise léxica* cria átomos (*tokens*) a partir do fluxo de caracteres e determina a sua classe e representação. Este módulo pode realizar alguma interpretação limitada de alguns átomos, por exemplo, para verificar se um identificador é um identificador de macro ou uma palavra reservada da linguagem de programação.

O *módulo de análise sintática* obtém o fluxo de átomos e produz uma estrutura de mais alto nível que representa o programa, denominada árvore sintática abstrata (*Abstract Syntax Tree* - AST). Uma AST é uma representação em árvore da estrutura sintática do código fonte de um programa. Cada nó da árvore representa uma construção que ocorre neste código.

O *módulo de tratamento de contexto* coleta informações de contexto de vários pontos do programa, e anota os resultados nos nós da AST. São exemplos de tratamento de contexto: relacionar informações de tipo de declarações com expressões; conectar coman-

[1] O *otimizador* apresentado na Figura 2 foi incorporado ao *back-end* na Figura 3.

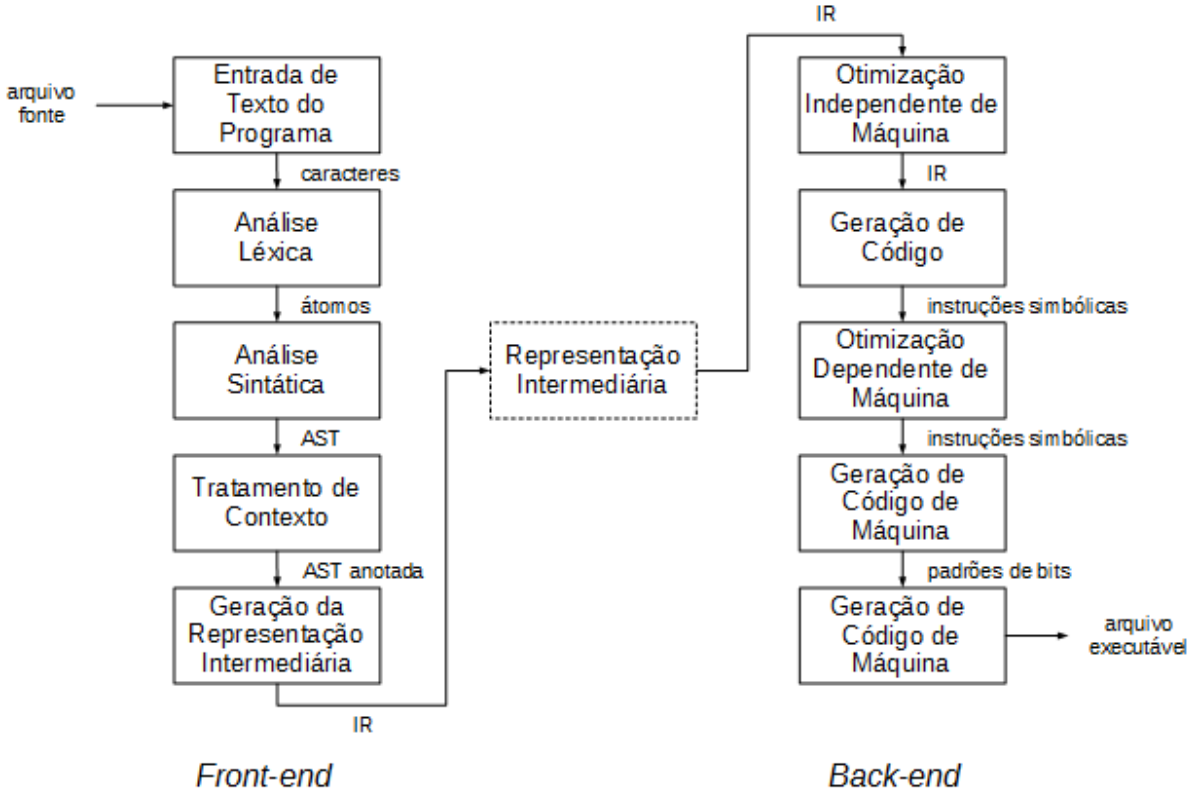

Figura 3: Módulos da estrutura de um compilador. Adaptada de (GRUNE et al., 2012).

dos *goto* a seus rótulos (*labels*); decidir quais chamadas de rotinas são locais e quais são remotas, em linguagens distribuídas. Essas anotações são então usadas para verificação de contexto ou são passadas aos módulos subsequentes para, por exemplo, auxiliar no processo de geração de código.

O *módulo de geração da representação intermediária* traduz as construções específicas da linguagem contidas na AST em construções mais gerais. Estas construções mais gerais constituem a IR.

O *módulo de otimização independente de máquina* realiza um pré-processamento na IR, aplicando transformações de otimização de código, com a intenção de melhorar a eficácia do módulo de geração de código.

O *módulo de geração de código* reescreve a AST em uma lista linear de instruções da máquina alvo, em uma forma mais ou menos simbólica. Ao final, ele seleciona instruções para segmentos da AST, alocação de registradores e organiza as instruções em uma ordem adequada.

O *módulo de otimização dependente de máquina* considera a lista de instruções simbólicas de máquina e tenta otimizá-la substituindo sequências de instruções de máquina por

sequências mais rápidas ou menores. Para isso, ele usa propriedades específicas da máquina alvo.

O *módulo de geração de código de máquina* converte as instruções simbólicas de máquina em um padrão de bits correspondente. Ele determina os endereços dos dados e código do programa e produz tabelas de constantes e tabelas de realocação.

O *módulo de saída do código executável* combina as instruções de máquina codificadas, as tabelas de constantes, as tabelas de realocação, e os *headers*, *trailers*, e outros materiais requeridos pelo sistema operacional em um arquivo de código executável.

2.3 Otimização de Código

Construções de linguagens de programação de alto nível podem introduzir um gasto substancial de tempo de execução se cada uma delas for traduzida diretamente em código de máquina. Para tratar de ineficiências deste tipo, os compiladores otimizadores possuem uma etapa de otimização de código.

Segundo Aho et al. (2006), otimização de código é a eliminação de instruções desnecessárias no código objeto, ou a substituição de uma sequência de instruções por uma sequência de instruções mais rápidas que tem o mesmo comportamento. A maioria dos compiladores comerciais possui otimizadores de código. Estes otimizadores geralmente proporcionam um bom nível de desempenho e, em alguns casos, o desempenho do código gerado está perto do pico de desempenho da máquina alvo. O uso clássico de otimização é para reduzir o tempo de execução do programa. Em outros contextos, o otimizador pode tentar reduzir o tamanho do código compilado ou outras propriedades, tal como, a energia que o processador consome executando o código (COOPER; TORCZON, 2011).

As transformações no código são guiadas usando um conjunto de otimizações. Usualmente, cada otimização consiste de um componente de análise, que identifica anomalias específicas de desempenho, e um componente de transformação, que elimina as anomalias através de uma série de transformações de código (JOSEPH et al., 2007).

As otimizações operam em diferentes granularidades ou escopos. Em geral, as transformações operam em um dos quatro escopos distintos: local, regional, global ou o programa todo (COOPER; TORCZON, 2011).

Transformações locais operam sobre um único bloco básico - uma sequência de comprimento máximo de código livre de ramificação. Elas estão entre as técnicas mais simples

que o compilador otimizador pode usar. A seguir, são listadas algumas das transformações que podem ser aplicadas a blocos básicos de código (AHO et al., 2006):

- Eliminação de *subexpressões locais comuns*, isto é, instruções que computam um valor que já foi computado;
- Eliminação de código morto (*dead code*), isto é, instruções que computam um valor que nunca é usado;
- Reordenação de instruções que não são dependentes; tal reordenação pode reduzir o tempo que um valor temporário precisa ser preservado em um registrador;
- Aplicação de leis algébricas para reordenar operandos em uma expressão e, talvez, simplificar a computação.

Transformações regionais operam sobre escopos maiores que um simples bloco básico, mas menores que uma subrotina. O compilador pode escolher regiões de diferentes maneiras. Uma região pode ser definida por alguma estrutura de controle do código fonte, tal como um laço encadeado. O compilador pode analisar o subconjunto de blocos básicos da região, formando um bloco básico estendido. O compilador pode ainda considerar um subconjunto do grafo do fluxo de controle definido por alguma propriedade teórica de grafo, tal como um denominador de relação. A seguir, são citadas duas transformações que podem ser aplicadas a regiões:

- Eliminação de *subexpressões superlocais comuns* atua sobre o escopo de um bloco básico estendido;
- *Loop Unrolling* replica o corpo de laços e ajusta a lógica que controla o número de iterações a serem executadas.

Transformações globais usam toda uma subrotina como contexto. A motivação para esse tipo de transformação é que decisões que são ótimas localmente podem ter consequências ruins em um contexto mais amplo. A seguir, são citadas duas transformações que podem ser aplicadas globalmente:

- Encontrar variáveis não inicializadas que são usadas pela subrotina;
- *Posicionamento de código global* usa informações de perfil obtidas da execução do código compilado para rearranjar a disposição do código executável.

Transformações relacionadas ao programa todo consideram escopos maiores que uma subrotina. Qualquer transformação que envolve mais de uma subrotina pode ser considerada desse tipo. A análise e otimizações ocorrem sobre o grafo de chamadas de subrotina do programa. A seguir, são citadas duas transformações que podem ser aplicadas neste contexto:

- *Substituição em linha* que substitui uma chamada de subrotina por uma cópia do corpo dessa subrotina.
- *Propagação de Constantes* que propaga informações sobre constantes em todo o programa.

Em geral, o processo de otimização do código é dividido em uma série de passos pelo otimizador. Cada passo obtém como entrada um código na forma de IR e produz uma versão reescrita deste código como saída. Esta estrutura permite que os passos sejam construídos e testados independentemente, e cria uma forma para o compilador fornecer diferentes níveis de otimização (COOPER; TORCZON, 2011). As transformações e a sequência em que elas são aplicadas pelo compilador no processo de otimização são descritas em um *plano de compilação*.

A escolha de transformações específicas e a sequência em que elas são aplicadas podem impactar na eficácia de um otimizador. Compiladores otimizadores clássicos fornecem diversos níveis de otimização (por exemplo, -O0, -O1, -O2, ...) que permitem ao usuário final tentar diferentes planos de compilação. Conforme apresentado na seção 1.1, pesquisas relacionadas à otimização de código em compiladores têm utilizado técnicas para criação de planos de compilação personalizados para cada programa (ou parte dele), selecionando o conjunto de transformações e a ordem em que elas são aplicadas.

2.4 Compiladores JIT

Códigos interpretados podem ser totalmente independentes do *hardware* e do sistema operacional em que executam. Entretanto, códigos interpretados são mais lentos que códigos compilados. Um meio para evitar a sobrecarga de interpretação é permitir ao interpretador gerar código de máquina para um segmento de código interpretado, antes desse segmento de código ser necessário. Isto é chamado de compilação *Just In Time* (JIT) (GRUNE et al., 2012).Uma vantagem dessa abordagem é que o processo de compilação

é ocultado do usuário, e que o código gerado pode ser adaptado para um processador particular.

Esta técnica é adequada somente se o processo de compilação gerar um ganho no desempenho em relação à execução do código interpretado. Modernas implementações JIT usam um número de técnicas para alcançar isto: os compiladores são cuidadosamente ajustados para serem tão rápidos quanto possível enquanto ainda estão produzindo códigos mais rápidos. Eles somente compilam partes do código que contribuem significantemente para o tempo de execução do programa; esses códigos são chamados de *hot spots* do programa. E alguns casos eles usam múltiplos níveis de qualidade de geração de código: um compilador rápido produz código de qualidade moderada para a maior parte do código, e um código mais sofisticado, porém de compilação mais lenta, para *hot spots* importantes (GRUNE et al., 2012).

Técnicas aplicadas em compiladores JIT utilizam informações coletadas em tempo de execução para guiar o processo de compilação. Essas técnicas são chamadas de adaptativas. A seguir, apresentam-se os componentes necessário em um compilador JIT adaptativo (ARNOLD et al., 2005):

Compilação Seletiva Apenas partes do programa são selecionadas para serem compiladas.

Feedback Directed Optimization Processo de coleta de informações, em tempo de execução, para guiar posteriormente o processo de compilação.

Feedback Directed Code Generation Processo de geração de código utilizando as informações coletadas em tempo de execução.

O processo de *Feedback Directed Optimization* coleta as informações sobre o tempo de execução utilizando técnicas de perfil (*profile*) de execução do programa. A seguir, são citadas algumas técnicas de perfil:

Contadores de Desempenho Contadores são inseridos no código para detectar os caminhos utilizados com mais frequência. Esses contadores são incrementados no momento em que determinada parte do programa é executada.

Amostragem O sistema coleta um subconjunto representativo de uma classe de eventos, permitindo um limitado gasto de tempo para se obter informações do perfil do programa.

Monitoração de Serviços Monitora os serviços requisitados pelo programa durante sua execução. São exemplos de serviços: a entrada e saída e o gerenciamento de memória.

Instrumentação Adição de código para coleta de informações.

Algumas transformações utilizadas na fase de otimização do compilador podem se beneficiar das informações sobre o tempo de execução do programa. Entre elas podem-se citar:

Inlining Consiste em replicar o código de uma rotina chamada no corpo da rotina que realiza a chamada, evitando o gasto de tempo na chamada da rotina, além de habilitar outras transformações.

Disposição de código Maximiza a localidade de instruções, colocando próximos os trechos de código mais prováveis de serem executados em sequência.

Agendamento de Instruções Maximiza o número de instruções que podem ser executadas paralelamente.

Múltiplas Versões Múltiplas versões da mesma parte do programa são geradas e ativadas dependendo de valores presentes no programa em tempo de execução.

2.5 Instrumentação de Código

Uma técnica utilizada para registrar o comportamento de um programa e medir o seu desempenho é o de inserir código em um programa e executar a sua versão modificada (BALL; LARUS, 1994). Código de instrumentação pode ser adicionado tanto de forma estática quanto de forma dinâmica. Na forma estática a instrumentação do código pode ser feita durante o processo de compilação do programa ou após a compilação, reescrevendo o código executável. Na forma dinâmica, o código do programa é modificado em tempo de execução.

Este trabalho utiliza apenas a forma dinâmica de instrumentação. Em programas Java, a instrumentação de código pode ser feita em tempo de execução diretamente em seus *bytecodes* através de arcabouços como, por exemplo, Javassist (CHIBA, 2000), ASM (KULESHOV, 2007) e BCEL (DAHM, 2001). Isso permite que cada método de um programa Java seja instrumentado para se obter o tempo de execução do método e também extrair seu perfil de execução (ARNOLD; HIND; RYDER, 2001).

2.6 Jikes RVM

A Jikes RVM (Research Virtual Machine) [2] é uma máquina virtual Java de código aberto escrita na linguagem de programação Java (ARNOLD et al., 2000), desenvolvida para auxiliar em pesquisas na área de máquinas virtuais. Ela possui um compilador JIT com diferentes níveis de otimização e também um sistema de compilação adaptativa.

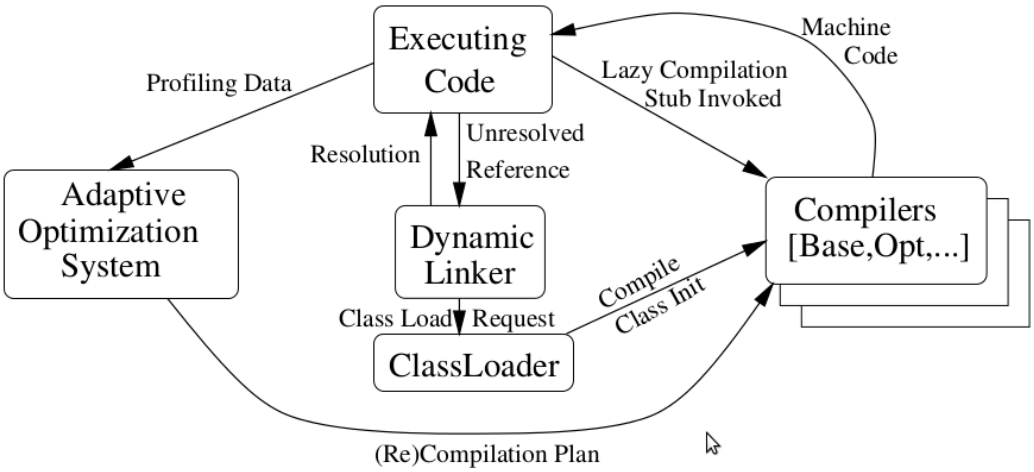

Figura 4: Cenários de compilação da Jikes RVM. Extraída de (ARNOLD et al., 2000).

A Figura 4 apresenta a estrutura dos cenários de compilação da Jikes RVM. Diferentemente de outras máquinas virtuais Java, a Jikes RVM não possui um interpretador de *bytecode*. Todos os métodos são compilados para código de máquina sob demanda. Ela possui dois tipos de compiladores: *baseline* e *optimizing*. O *baseline* é um compilador mais rápido que não aplica nenhum tipo de otimização. O *optimizing* aplica uma ou mais transformações de otimização de código. As otimizações estão agrupadas em diferentes níveis de otimização (O0, O1, O2) e podem ser habilitadas ou desabilitadas individualmente, via linha de comando. A ordem de aplicação das transformações é fixa. A Tabela 1 apresenta a descrição das otimizações contidas na Jikes RVM versão 3.1.4. Os dados dessa tabela foram extraídos do *help* da Jikes RVM.

O sistema de compilação adaptativa da Jikes RVM utiliza os dois tipos de compiladores. Ele tenta encontrar um bom balanceamento entre tempo de compilação e tempo de execução. Inicialmente, os métodos são compilados usando o compilador *baseline*, que é rápido, porém produz um código de máquina mais lento. Ao longo da execução do programa, os métodos que são executados frequentemente, denominados *hot spots*, são identificados e são recompilados pelo compilador *optimizing*, que é mais lento e complexo,

[2] http://www.jikesrvm.org/

Tabela 1: Otimizações da Jikes RVM. Extraídas do *help* da Jikes RVM versão 3.1.4.

Otimização	**Nível**	**Descrição**
field_analysis	0	Eagerly compute method summaries for flow-insensitive field analysis
inline	0	Inline statically resolvable calls
inline_guarded	0	Guarded inlining of non-final virtual calls
inline_guarded_interfaces	0	Speculatively inline non-final interface calls
inline_preex	0	Pre-existence based inlining
local_constant_prop	0	Perform local constant propagation
local_copy_prop	0	Perform local copy propagation
local_cse	0	Perform local common subexpression elimination
control_static_splitting	1	CFG splitting to create hot traces based on static heuristics
escape_scalar_replace_aggregates	1	If possible turn aggregates (objects) into variable definition/uses
escape_monitor_removal	1	Try to remove unnecessary monitor operations
reorder_code	0	Reorder basic blocks for improved locality and branch prediction
reorder_code_ph	1	Reorder basic blocks using Pettis and Hansen Algo2
h2l_inline_new	0	Inline allocation of scalars and arrays
h2l_inline_write_barrier	1	Inline write barriers for generational collectors
h2l_inline_primitive_write_barrier	1	Inline primitive write barriers for certain collectors
regalloc_coalesce_moves	0	Attempt to coalesce to eliminate register moves?
regalloc_coalesce_spills	0	Attempt to coalesce stack locations?
osr_guarded_inlining	1	Insert OSR point at off branch of guarded inlining?
osr_inline_policy	1	Use OSR knowledge to drive more aggressive inlining?
l2m_handler_liveness	2	Store liveness for handlers to improve dependence graph at PEIs

porém gera um código de máquina com mais qualidade e eficiência.

Este trabalho utiliza a Jikes RVM como máquina virtual para a execução dos programas e estruturação do arcabouço. Decidiu-se por utilizar essa JVM por ela ser de código aberto e permitir modificações em seu código fonte utilizando a linguagem Java. Ela possui diferentes tipos de otimizações implementadas, o que facilita a realização das pesquisas experimentais.

2.7 Conclusão

A maioria dos compiladores comerciais possuem otimizadores de código que aplicam

transformações no código do programa na tentativa de gerar um código objeto de melhor qualidade. Em geral, heurísticas definidas manualmente pelos desenvolvedores do compilador são usadas para guiar o processo de transformações do código para um determinado objetivo. Este processo é caro, consome muito tempo e é propenso a erros, e pode levar a um desempenho sub-ótimo. Para lidar com tais questões, algoritmos de aprendizado de máquina tem sido usado para automatizar o processo de geração de heurísticas em compiladores. O próximo capitulo apresenta os principais conceitos sobre aprendizado de máquina que são utilizados neste trabalho.

3 APRENDIZADO DE MÁQUINA

Aprendizado de máquina pode ser definido como um conjunto de métodos que podem automaticamente detectar padrões em dados e então, utilizar padrões desconhecidos para predizer dados futuros, ou executar outros tipos de tomadas de decisão sob incerteza (MURPHY, 2012). Algoritmos de aprendizado de máquina são usualmente divididos em três categorias principais: supervisionados, não-supervisionados e por reforço.

O aprendizado de máquina supervisionado consiste em aprender uma função a partir de dados de treinamento (MITCHELL, 1997). Os dados de treinamento são constituídos de vetores de entradas juntamente com seus vetores de saídas correspondentes (BISHOP, 2006). Utilizando os dados de treinamento, um algoritmo de aprendizado supervisionado produz um modelo capaz de predizer saídas a partir de entradas não contidas nos dados de treinamento.

O aprendizado de máquina não-supervisionado consiste em encontrar padrões de interesse em dados, a partir somente de dados de entrada, sem nenhuma correspondência com dados de saída (MURPHY, 2012). O objetivo em problemas de aprendizado supervisionado pode ser descobrir grupos de exemplos similares dentro dos dados, denominado agrupamento (*clustering*), ou determinar a distribuição de dados dentro do espaço de entradas, conhecido como estimativa de densidade (*density estimation*) (BISHOP, 2006).

O aprendizado por reforço consiste em aprender a partir da interação para alcançar um objetivo. O aprendizado é realizado por um agente aprendiz através de sua interação com um ambiente. O agente aprende de maneira autônoma uma política ótima de atuação. Diferentemente de outras formas de aprendizado de máquina, o agente não é ensinado por meio de exemplos fornecidos por um supervisor. Ele deve aprender ativamente, através de um processo de tentativa e erro (SUTTON; BARTO, 2017).

3.1 Aprendizado de Máquina Supervisionado

No aprendizado de máquina supervisionado, o objetivo é aprender uma mapeamento de entradas x para saídas y, dado um conjunto rotulado de pares entrada-saída $D = \{(x_i, y_i)\}_{i=1}^{N}$. Onde, D é denominado de conjunto de *treinamento*, e N é o número de exemplos de treinamento (MURPHY, 2012).

Cada entrada x_i de treinamento é um vetor de números denominado características ou atributos. Entretanto, x_i pode ser um objeto de estrutura mais complexa como, por exemplo, uma imagem, uma sentença, uma mensagem de email, etc.

Os algoritmos de aprendizado de máquina são divididos a partir do tipo de saída. Basicamente, existem dois tipos de algoritmos: *classificação* e *regressão*. Nos algoritmos do tipo classificação, as saídas são definidas por valores discretos. Cada valor de saída é denominado classe ou rótulo. Nos algoritmos do tipo regressão, as saídas são definidas por valores contínuos.

O objetivo dos algoritmos de classificação é aprender um mapeamento de entradas x em saídas y, onde $y \in \{1, ..., C\}$, com C sendo o número de classes. Se $C = 2$, denomina-se classificação binária; se $C > 2$, denomina-se classificação *multiclasse*. Se os rótulos das classes não são mutuamente exclusivos, denomina-se classificação *multirótulos* (MURPHY, 2012).

Uma maneira de formalizar o problema de classificação é com uma função de aproximação. Assume-se $y = f(x)$ para alguma função f desconhecida, e o objetivo do aprendizado é estimar a função f dado um conjunto de treinamento rotulado, e então predizer usando $\hat{y} = \hat{f}(x)$. O principal objetivo é predizer sobre novas entradas, ou seja, aquelas que não tenham sido vistas anteriormente (MURPHY, 2012).

O tipo de aprendizado supervisionado utilizado neste trabalho é o de classificação de valores, utilizado para predizer as saídas 0 ou 1. Entretanto, algoritmos de aprendizado utilizando regressão também podem ser utilizados. Para isso, aplica-se uma transformação dos valores de saída para as categorias 0 ou 1.

3.1.1 Algoritmos

Esta seção apresenta uma breve descrição dos algoritmos de aprendizado de máquina supervisionado que podem ser utilizados com o arcabouço `ML4JIT`.

Árvores de Decisão Cria um modelo que prediz o valor de uma variável de saída, apren-

dendo regras de decisão simples inferidas a partir dos dados de características (MITCHELL, 1997).

Regressão Logística Apesar do nome, é um modelo linear para classificação. Neste modelo, as probabilidades utilizadas para descrever as possíveis saídas de uma única tentativa são modeladas usando uma função logística (BISHOP, 2006).

Support Vector Machine São modelos de aprendizado estatístico que buscam separar os exemplos de treinamento através da maximização de uma margem entre os exemplos mais próximos de cada classe (MURPHY, 2012).

Nearest Neighbors É um tipo de aprendizado baseado em instância ou aprendizado não-generalizante. Ele não tenta construir um modelo interno geral, mas simplesmente armazena instâncias dos dados de treinamento. A classificação é calculada a partir de um voto de maioria simples dos vizinhos mais próximos de cada ponto: um ponto de consulta é atribuído à classe de dados que tem mais representantes dentro dos vizinhos mais próximos do ponto (ERTEL, 2011).

Random Forest É um tipo classificador que utiliza diversos modelos simples para fazer a classificação, com o objetivo de aumentar a acurácia da previsão. A ideia do *Random Forest* é construir uma floresta de *Árvores de Decisão* e utilizar todas as árvores para classificar uma nova instância. Cada árvore é gerada utilizando um subconjunto das variáveis disponíveis escolhidas de forma aleatória, além de um diferente conjunto de dados, gerado pela técnica de amostragem *bootstrap* (BREIMAN, 2001).

3.2 Aprendizado Ativo

O aprendizado ativo (*active learning*) (SETTLES, 2012) é um tipo especial de aprendizado de máquina semi-supervisionado na qual um algoritmo aprendiz é capaz de interativamente perguntar ao usuário (ou a alguma outra fonte de informações) para obter as saídas desejadas em novos pontos de dados.

Diferentemente do aprendizado de máquina supervisionado, que utilizam para predição os dados obtidos através do treinamento do algoritmo, o aprendizado ativo desenvolve e testa novas hipóteses como parte de um processo contínuo e interativo de aprendizagem.

A Figura 5 apresenta um diagrama descrevendo o processo de um tipo de aprendizado ativo, denominado *pool-based* (SETTLES, 2012). Nesse processo, uma pequena amostra

rotulada é fornecida a um algoritmo de aprendizado semi-supervisionado, por meio de um treinamento. Após esse treinamento o algoritmo de aprendizado ativo analisa os dados que ainda não foram rotulados, e busca entre eles aquele que fornecerá maior ganho de informação para o modelo. Esse dado é então enviado a um especialista para rotulá-lo e então ele é incorporado ao conjunto de amostras rotuladas.

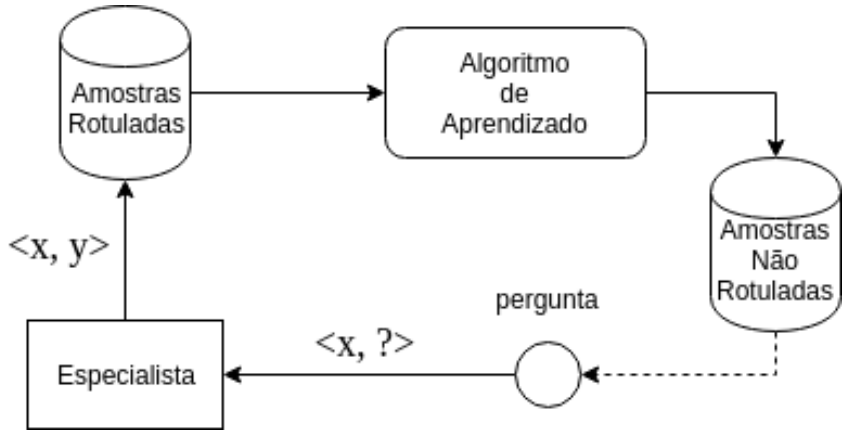

Figura 5: Processo de aprendizado ativo *pool-based*. Adaptada de (SETTLES, 2012).

O uso do processo de aprendizado ativo apresentado pode ser adaptado para o uso em compiladores. Ao invés de se perguntar a um especialista para se obter as saídas desejadas, pode-se inferir a saída sobre o conjunto de otimizações aplicado através da coleta do perfil de execução do programa ou parte dele (funções ou métodos).

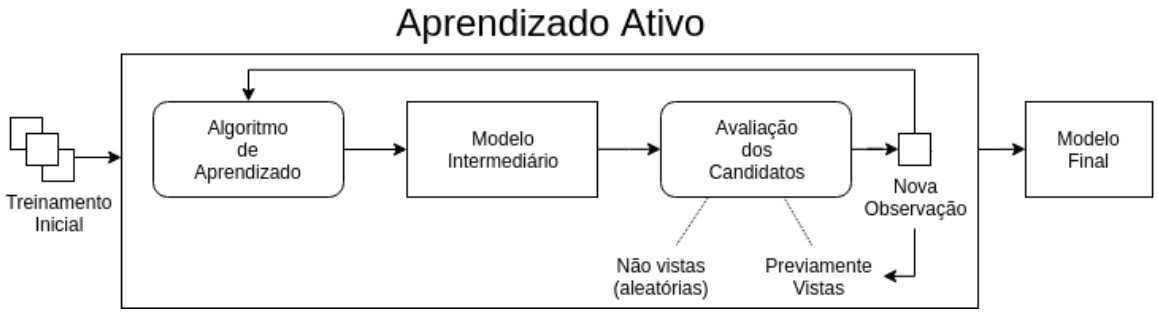

Figura 6: Processo adaptado de aprendizado ativo para uso em compiladores. Adaptada de (OGILVIE et al., 2017).

A Figura 6 apresenta uma proposta de adaptação do processo de aprendizado ativo para uso em compiladores (OGILVIE et al., 2017). Uma pequena amostra é fornecida a um algoritmo de aprendizado, por meio de um treinamento, gerando um modelo intermediário. Durante o processo de aprendizado, novas amostras não fornecidas no treinamento são aplicadas e o resultado de sua aplicação é analisado. Caso se verifique que houve um ganho no desempenho com a aplicação da amostra, ela é fornecida ao algoritmo de aprendizado,

que gera um novo modelo intermediário. Esse processo é repetido até que se atinja algum critério de finalização.

3.3 Aprendizado por Reforço

Aprendizado por reforço é uma forma de aprendizado de máquina para o problema de aprender a partir da interação para alcançar um objetivo. O aprendizado é realizado por um agente aprendiz através de sua interação com um ambiente. O agente aprende de maneira autônoma uma política ótima de atuação. Diferentemente de outras formas de aprendizado de máquina, o agente não é ensinado por meio de exemplos fornecidos por um supervisor. Ele deve aprender ativamente, através de um processo de tentativa e erro (SUTTON; BARTO, 2017).

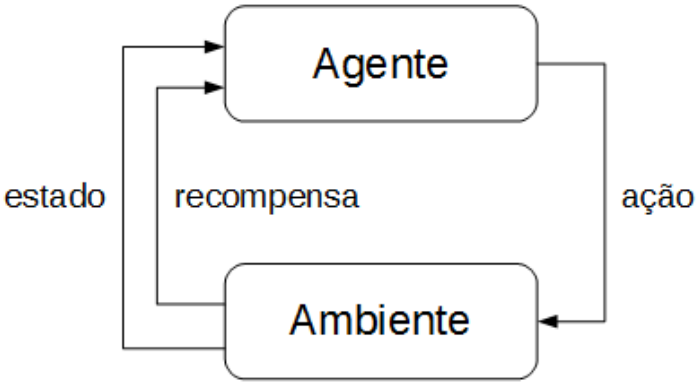

Figura 7: A interação entre o agente e o ambiente em aprendizado por reforço.

A Figura 7 apresenta a interação entre um agente e um ambiente. O ambiente compreende tudo que é externo ao agente. O agente interage continuamente com o ambiente em intervalos de tempo discreto. O agente observa o estado corrente s_t do ambiente e seleciona uma ação a_t para ser realizada. Ao executar essa ação a_t (que altera o estado do ambiente) o agente recebe uma recompensa numérica $r_{s,a}$, que indica quão desejável é o estado resultante s_{t+1}. Assim, o agente pode determinar, após várias interações com o ambiente, qual a melhor ação a ser executada em cada estado, isto é, a melhor política de atuação.

O objetivo do agente é aprender uma política ótima de atuação que maximize o total de recompensas numéricas recebidas do ambiente ao longo de sua execução, independentemente do estado inicial. Esse problema pode ser modelado como um Processo Markoviano de Decisão (KAELBLING; LITTMAN; MOORE, 1996; MITCHELL, 1997).

3.3.1 Elementos do Aprendizado por Reforço

Além do agente e do ambiente, pode-se identificar quatro sub-elementos principais de um sistema de aprendizado por reforço: uma *política*, uma *função de recompensa*, uma *função valor*, e, opcionalmente, um *modelo* do ambiente (SUTTON; BARTO, 2017).

Uma *política* define a forma como o agente comporta-se em um dado momento. Ela é um mapeamento dos estados percebidos do ambiente para ações a serem tomadas quando se está naquele estado, $\pi : \mathcal{S} \rightarrow \mathcal{A}$. A política define a ação a ser tomada em cada estado s_t: $a_t = \pi(s_t)$ (ALPAYDIN, 2010). O agente precisa aprender uma política ótima (ou próximo da ótima) de modo a realizar o seu objetivo. Em geral, políticas podem ser estocásticas.

Uma *função de recompensa* define o objetivo em um problema de aprendizado por reforço. Ela mapeia cada estado percebido (ou o par estado-ação) do ambiente a um único número, uma *recompensa*, indicando o quão desejável é aquele estado. A tarefa do agente é maximizar o total de recompensas recebidas ao longo do tempo. A função de recompensa define quais os eventos são bons ou ruins para o agente. Em geral, funções de recompensa são estocásticas.

Uma *função valor*, $V^{\pi}(s_t)$, determina o *valor* da recompensa esperada acumulada que o agente irá receber enquanto seguir a política π, iniciando do estado s_t. Em modelos de aprendizado por reforço em que as tarefas são contínuas, denominados modelos de *horizonte infinito*, calcula-se o valor de um estado utilizando um fator de desconto:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{3.1}$$

onde,

- r_{t+i} é a sequência de recompensas recebidas a partir do estado s_t, usando a política π.
- γ é o fator de desconto, com $0 \leq \gamma < 1$.

O objetivo do agente é encontrar uma política ótima, π^*, tal que (ALPAYDIN, 2010):

$$V^*(s_t) = \max_{\pi} V^{\pi}(s_t), \forall s_t \tag{3.2}$$

Um *modelo* do ambiente é algo que simula o comportamento daquele ambiente. Por exemplo, dado um estado e uma ação o modelo pode prever a recompensa e o próximo

estado. Ele é usado para planejamento, através do qual se tem um meio de decidir o curso de uma ação considerando situações futuras antes de realmente experimentá-las.

3.4 Aprendizado Dinâmico

Este trabalho utiliza o termo *aprendizado dinâmico* para indicar de forma abstrata que o tipo de algoritmo de aprendizado de máquina utilizado permite um processo de aprendizagem incremental;isto é, ao invés de se obter os dados para treinamento do modelo de aprendizado de forma estática, executando uma busca exaustiva ou aleatória sobre o espaço de otimizações, como ocorre no aprendizado supervisionado, esses dados podem ser obtidos em tempo de execução do programa que se deseja analisar. A realização do processo de aprendizado dinâmico proposto nesse trabalho pode ser feita tanto por algoritmos de aprendizado ativo (SETTLES, 2012) como por algoritmos de aprendizado por reforço (SUTTON, 1996).

3.5 Conclusão

Os algoritmos de aprendizado de máquina podem ser utilizados em compiladores para auxiliar no processo de descoberta do melhor conjunto de otimizações a ser aplicado em um programa ou específico para cada unidade do programa (funções ou métodos). Para isso, são extraídas características para descrever o programa ou cada uma de suas unidades. Essas características são usadas tanto na fase de treinamento quanto na fase de teste, em que o algoritmo faz a predição das saídas. O próximo capítulo apresenta um arcabouço para pesquisa em compiladores que utiliza os algoritmos de aprendizado de máquina apresentados para a descoberta do melhor conjunto de otimizações para cada método de um programa Java.

4 DESCRIÇÃO DO ARCABOUÇO

Este capítulo apresenta uma visão geral da estrutura e das funcionalidades do `ML4JIT`, um arcabouço para pesquisa com aprendizado de máquina em compiladores JIT para a linguagem de programação Java. Ele foi projetado para possibilitar a realização de pesquisas experimentais com a intenção de se descobrir o melhor conjunto de otimizações específico para cada um dos métodos de um programa Java, na tentativa de reduzir o tempo de execução do programa. O processo de descoberta é auxiliado pelo uso de algoritmos de aprendizado de máquina.

O arcabouço possibilita ao pesquisador realizar experimentos controlados para a descoberta do melhor conjunto de otimizações específico para cada método de um programa utilizando diferentes níveis de otimização de um compilador JIT. Ele permite ainda o uso de diferentes tipos de algoritmos de aprendizado de máquina, assim como, a utilização de parâmetros de configuração distintos para o mesmo algoritmo de aprendizado.

4.1 Motivação

Conforme apresentado na seção 1.1, técnicas de aprendizado de máquina têm sido utilizadas em compiladores para o desenvolvimento de heurísticas de otimização de código (STEPHENSON et al., 2003; CAVAZOS; MOSS, 2004; FURSIN et al., 2011). Em especial, pesquisa têm sido realizadas para se tentar descobrir o melhor conjunto de otimizações específico para cada um dos métodos de um programa. Cavazos e O'boyle (2006) aplicaram a técnica de *logistic regression* para determinar automaticamente quais otimizações são melhores para cada método de um programa desenvolvido em linguagem Java. Como avanço desse trabalho, Sanchez et al. (2011) utilizaram a técnica de *Support Vector Machine* (SVM) e aplicaram em uma máquina virtual de grande porte, a Testarossa da IBM.

Entretanto, a tentativa de reprodução de alguns trabalhos apontou diversos desafios e dificuldades. Em geral, os autores utilizam uma máquina virtual Java específica e

a modificam para possibilitar a realização dos experimentos. Essas modificações nem sempre estão disponíveis ou mesmo detalhadas para permitir a reprodução do ambiente de pesquisa. Há casos ainda em que a JVM utilizada é proprietária e seu código fonte não é aberto, impossibilitando que ela seja utilizada facilmente por pesquisadores.

Outro ponto a ser considerado é que boa parte dos trabalhos apresentam resultados de experimentos realizados com um único tipo de algoritmos de aprendizado de máquina. Com isso, não se sabe ao certo se outros tipos de algoritmos seriam mais adequados ao problema que se está tratando.

Por esses motivos, decidiu-se desenvolver um arcabouço de código aberto e livre para permitir a realização de pesquisas com otimização de código em compiladores JIT para a linguagem Java com o auxilio de aprendizado de máquina. Definiu-se que o arcabouço deve ser extensível, permitindo que novos tipos de experimentos possam ser acoplados em sua estrutura. A próxima seção apresenta os principais requisitos que direcionaram o desenvolvimento do `ML4JIT`.

4.2 Requisitos

A proposta base do `ML4JIT` é que ele seja um arcabouço de código aberto e livre e tenha uma estrutura extensível para permitir que novos tipos de experimentos sejam acoplados facilmente. Um conjunto de requisitos direcionaram o seu desenvolvimento. A seguir, são apresentados os principais deles. Eles estão divididos em requisitos funcionais e requisitos não funcionais.

Requisitos Funcionais

RF01 - Permitir a instrumentação de métodos de um programa Java sem a necessidade do seu código fonte e em tempo de execução do programa. Esse requisito possibilita que o *bytecode* dos métodos de um programa Java sejam instrumentados para permitir que dados, como o tempo de execução de cada método, sejam coletados em tempo de execução do programa.

RF02 - Extrair e registrar as características dos métodos de forma estática ou dinâmica. Esse requisito define que o arcabouço deve ser capaz de extrair características de cada um dos métodos que são utilizadas para descrevê-los e também para serem usadas nos algoritmos de aprendizado de máquina.

RF03 - Registrar o *bytecode* dos métodos após a transformação do código. Esse

requisito permite ao pesquisador verificar se o código sofreu a transformação adequada.

RF04 - Registrar a quantidade de vezes que os métodos do programa são chamados em tempo de execução. Com esses dados, o pesquisador pode fazer uma análise prévia do programa e verificar quais são os métodos que mais são chamados durante a execução.

RF05 - Extrair dados para a etapa de treinamento do processo de aprendizado de máquina. Esse requisito define que o arcabouço deve possuir recursos para controlar o processo de coleta de dados que serão usados na etapa de treinamento dos algoritmos de aprendizado de máquina. Dentre os dados coletados, deve-se obter as características e o tempo de execução de cada método do programa.

RF06 - Acionar os algoritmos de aprendizado de máquina e coletar os dados de predição para cada método do programa, tanto de forma estática quanto dinâmica. Este requisito define que o arcabouço deve possuir recursos para acionar os algoritmos de aprendizado de máquina e coletar os dados de predição. Para o algoritmo de aprendizado de máquina devem ser informadas as características do método e como resultado ele deve retornar o conjunto de otimizações que deve ser aplicado ao método.

Requisitos Não Funcionais

RNF01 - Ser o mais independente possível de uma máquina virtual Java específica. O objetivo desse requisito é que poucas alterações sejam feitas na máquina virtual Java, criando assim uma independência de uma implementação específica.

RNF02 - Permitir o acoplamento de novos recursos de instrumentação de código. Esse requisito define que o arcabouço deve ter uma arquitetura que permita que novos tipos de pesquisas experimentais possam ser acopladas como, por exemplo, para se tentar reduzir o consumo de energia de um programa, por meio do conjunto de otimizações a ser aplicada a cada método do programa.

RNF03 - Permitir o uso de diferentes tipos de algoritmos de aprendizado de máquina e também o uso de parâmetros de configuração distintos para cada algoritmo. Com este requisito pretende-se que o pesquisador possa comparar diferentes tipos de algoritmos de aprendizado de máquina para a descoberta do melhor conjunto de otimizações a serem aplicadas de forma específica para cada método do programa.

Para o funcionamento adequado do arcabouço, a máquina virtual Java que será utilizada nas pesquisas experimentais deve possuir também um conjunto de funcionalidades. Essas funcionalidades são descritas na seção 4.5.

4.3 Estrutura do Arcabouço

Esta seção apresenta uma visão geral da estrutura do arcabouço `ML4JIT`. Seu objetivo principal é o de possibilitar a realização de pesquisas experimentais com a intenção de se descobrir o melhor conjunto de otimizações específico para cada um dos métodos de um programa Java, na tentativa de reduzir o tempo de execução do programa. Algoritmos de aprendizado de máquina são utilizados no processo de descoberta dos conjuntos de otimizações.

Uma etapa inicial para o uso de aprendizado de máquina é a de treinamento do modelo de aprendizado. Nesta etapa, são utilizados diversos programas com o objetivo de se obter o tempo de execução de cada um de seus métodos, testando-se diferentes planos de otimização. Como resultado, obtém-se quais os melhores planos de otimização para cada um dos métodos analisados e cria-se um modelo de aprendizado capaz de predizer qual é o melhor conjunto de otimizações específico para cada método de um programa que não participou da etapa de treinamento, isto é, métodos não vistos pelo algoritmo de aprendizado de máquina.

Os dados utilizados nos algoritmos de aprendizado de máquina são extraídos diretamente do programa. Através de uma análise estática do *bytecode* dos métodos, se extraem as características que descrevem aquele método. Elas são utilizadas para a criação do modelo de aprendizado e também como dados de entrada para a predição que é realizada pelo algoritmo.

Os demais dados utilizados para o treinamento do modelo de aprendizado, como o tempo de execução e de compilação dos métodos, são obtidos de forma dinâmica, em tempo de execução do programa. O tempo de compilação de cada método deve ser fornecido pela máquina virtual. O tempo de execução de cada método deve ser obtido através de dados coletados do seu perfil de execução. Os dados de perfil são obtidos através da instrumentação do *bytecode* do método.

O arcabouço é dividido em três componentes principais: uma máquina virtual Java (JVM), um *agente externo* e um conjunto de *scripts*. A Figura 8 apresenta a relação existente entre os componentes do `ML4JIT`.

A máquina virtual Java é responsável pela execução dos programas. Ela deve possuir algumas funcionalidades, necessárias para a utilização do arcabouço. Essas funcionalidades são detalhadas na seção 4.5.

O agente externo é acoplado e acionado pela JVM e tem a responsabilidade de analisar

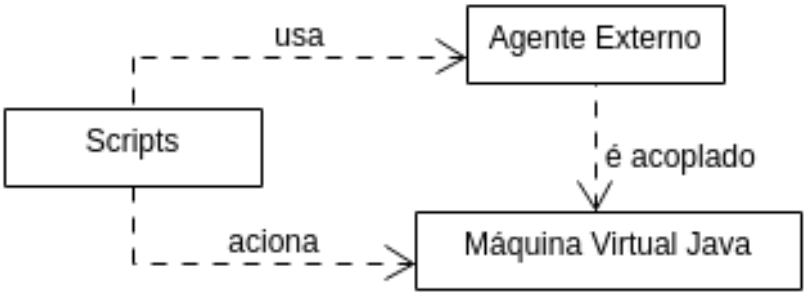

Figura 8: Relação entre os componentes do arcabouço `ML4JIT`.

e instrumentar o *bytecode* dos métodos do programa, em tempo de execução. A utilização de um agente externo possibilita que o processo de análise e instrumentação de código seja independente de uma JVM específica. Com isso, o arcabouço pode ser utilizado em diferentes máquinas virtuais, desde que elas possuam as funcionalidades necessárias.

Os *scripts* são utilizados para o controle de execução dos recursos do arcabouço. Eles foram desenvolvidos para facilitar a criação dos comandos adequados para a execução de um determinado recurso e também para o processamento dos dados obtidos com a execução.

O `ML4JIT` é constituído por um conjunto de *recursos* que engendram a utilização dos componentes na definição de experimentos descritos pelo usuário. A Figura 9 apresenta os principais recursos disponíveis no arcabouço. Esses recursos são: `Counter`, `Features Extractor`, `Training` e `Machine Learning`. As próximas seções detalham cada um deles.

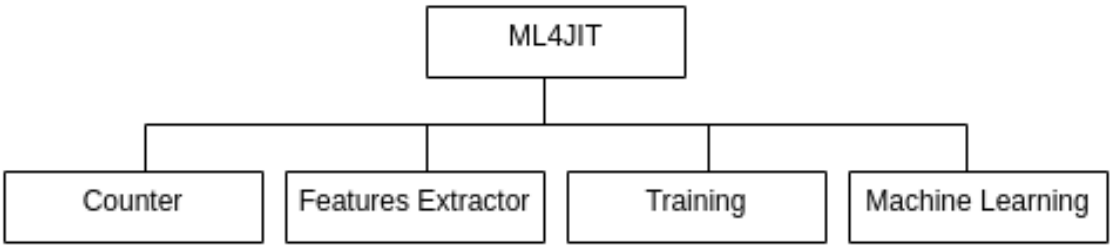

Figura 9: Principais recursos disponíveis no `ML4JIT`.

4.3.1 Counter

O `Counter` é um recurso que realiza a contagem de quantas vezes cada método do programa é chamado durante a execução. Os dados resultantes podem ser utilizados para

uma análise preliminar do programa e também na descoberta dos métodos que mais são acionados.

O processo de contagem é realizado acoplando-se o *agente externo* à JVM no modo `counter`. O agente é responsável por instrumentar o *bytecode* de cada método do programa, adicionando instruções antes de todas as outras, que registram que o método foi chamado.

Como resultado, o agente gera um arquivo texto com os dados coletados em tempo de execução. O cabeçalho do arquivo contém a quantidade de métodos analisados e a quantidade total de chamadas dos métodos. Cada registro do arquivo contém: o *id* do método, a assinatura do método e a quantidade de vezes que o método foi chamado.

O `Counter` é utilizado com a execução do *script* `counter`. Esse *script* recebe como argumento o identificador de um *benchmark* pré-configurado no arcabouço e também o nome do programa a ser analisado e, aciona a JVM para a execução do programa.

4.3.2 Features Extractor

O `Features Extractor` é um recurso que analisa o *bytecode* dos métodos do programa e, para cada método, gera um vetor de características $\overrightarrow{F}$ que o descreve. Esse vetor deve conter informações suficientes sobre o método para permitir ao modelo de aprendizado de máquina correlacionar essas informações com um plano de otimização que resultam em um bom desempenho. Entretanto, determinar tais informações é uma tarefa difícil. Decidiu-se selecionar características que são simples de serem calculadas através da análise do *bytecode* do método em um único passo.

A Tabela 2 apresenta um conjunto de 24 características utilizadas, no arcabouço, para descrever um método. Elas foram definidas a partir do trabalho de Cavazos e O'boyle (2006). O valor de cada característica será uma entrada do vetor $\overrightarrow{F}$. As duas primeiras entradas são valores inteiros definindo o tamanho de código e de dados do método. As próximas seis entradas são propriedades booleanas simples (representadas por 0 ou 1) do método. As demais entradas são a fração de *bytecodes* pertencentes a uma determinada categoria.

Esse conjunto não pretende ser completo, novas características podem ser identificadas e incorporadas ao arcabouço. É possível também utilizar apenas um subconjunto das características propostas, permitindo assim que sejam realizadas pesquisas com diferentes subconjuntos na tentativa de se descobrir quais deles são as mais relevantes para se definir

Tabela 2: Conjunto de características utilizadas para descrever um método.

Característica	**Significado**
bytecodes	Número de *bytecodes* no método
locals	Número máximo de palavras alocadas para variáveis locais
synch	O método é `synchronized`
exceptions	O método tem código de tratamento de exceção
leaf	O método não contém chamada a outros métodos
final	O método é declarado como `final`
private	O método é declarado como `private`
static	O método é declarado como `static`
array_load	Fração de *bytecodes* que são de leitura em *array*
array_store	Fração de *bytecodes* que são de escrita em *array*
arithmetic	Fração de *bytecodes* que são de operação aritmética
compare	Fração de *bytecodes* que são de comparações
branch	Fração de *bytecodes* que são de ramificação
switch	Fração de *bytecodes* que são um `switch`
put	Fração de *bytecodes* que são um `put`
get	Fração de *bytecodes* que são um `get`
invoke	Fração de *bytecodes* que são um `invoke`
new	Fração de *bytecodes* que são um `new`
array_length	Fração de *bytecodes* que são um `ArrayLength`
athrow	Fração de *bytecodes* que são um `athrow`
checkcast	Fração de *bytecodes* que são um `checkcast`
monitor	Fração de *bytecodes* que são um `monitor`
multi_newarray	Fração de *bytecodes* que são um `Multi Newarray`
conversion	Fração de *bytecodes* que são de conversão de tipos primitivos

o melhor plano de otimização para um método.

Os vetores de características dos métodos são utilizados tanto na etapa de treinamento quanto na etapa de avaliação e testes. Na etapa de treinamento, eles são associados aos melhores planos de otimização definidos para o método. Na etapa de avaliação e testes, eles são usados como dado de entrada para o algoritmo de aprendizado de máquina predizer o melhor plano de otimização para o método.

A geração do vetor de características para cada método do programa pode ser feita tanto de forma dinâmica como de forma estática. Na forma dinâmica, o agente externo é responsável por extrair as características de cada um dos métodos em tempo de execução. Entretanto, como isso gera uma piora no desempenho do programa, é possível extrair as características de forma estática, e utilizá-las posteriormente na etapa de treinamento e de avaliação e testes.

A extração de características de forma estática é feita executando o *script* `features`. Esse *script* recebe como argumento o identificador de um *benchmark* pré-configurado no

arcabouço e aciona o programa `FeatureExtractor`, que gera um arquivo em que cada registro representa o vetor de característica de um método do programa. Esse arquivo deve então, na etapa de treinamento, ser informado ao *agente externo*, através do parâmetro de configuração `features`, descrito na seção 4.3.6. Na etapa de avaliação e testes, o arquivo deve ser informado à máquina virtual, por meio de um parâmetro de configuração.

4.3.3 Training

Uma etapa importante no uso de aprendizado de máquina supervisionado é a de treinamento do modelo. Nesta etapa, são coletados exemplos de entradas e de saídas desejadas, para que os algoritmos aprendam uma regra geral que mapeia entradas e saídas.

O `Training` é um recurso que coleta os dados para treinamento do algoritmo de aprendizado de máquina. Esses dados devem identificar quais opções de otimização são as melhores para cada um dos métodos do programa.

A coleta dos dados de treinamento inicia com a seleção do nível de otimização, de uma JVM, que se quer analisar. O programa é então executado diversas vezes, e para cada uma delas, um conjunto de otimizações diferente é aplicado aos métodos do programa. O conjunto de otimizações é definido habilitando ou desabilitando a aplicação das otimizações pertencentes ao um determinado nível. A ordem de aplicação das otimizações é definida pela JVM e não é alterada.

O arcabouço permite que para um determinado nível de otimização sejam obtidos dados de todas as permutações de otimizações possíveis ou, dados de um conjunto de N permutações, selecionadas aleatoriamente. Entretanto, sempre uma execução do programa é feita com todas as opções de um determinado nível de otimização habilitadas. Os dados obtidos com essa execução são usados como referência para a comparação com as demais execuções.

A cada execução do programa são registrados, para cada método do programa, o tempo de compilação dinâmica e uma lista de tempos de execução. Cada dado da lista refere-se ao tempo de execução de uma chamada do método. A partir dessa lista, é possível definir o valor do tempo de execução do método que deve ser utilizado. O arcabouço permite definir esse valor pela média ou pela mediana dos valores da lista.

Ao final de todas as execuções são selecionados, para cada método do programa, os conjuntos de otimizações que obtiveram uma melhora de tempo em relação ao percentual limite configurado no arcabouço, sempre considerando a execução de referência. O tempo

utilizado para comparação pode ser obtido apenas com o valor do tempo de execução do método ou, pela soma do valor do tempo de execução do método com o seu tempo de compilação dinâmica.

Os conjuntos de otimização selecionados por método são associados ao vetor de características daquele método. Esses dados são registrados em uma tabela e gravados em um arquivo para que sejam utilizados para o treinamento do algoritmo de aprendizado de máquina.

Embora o processo de coleta de dados de treinamento opere conforme descrito, ele pode ser adaptado para a utilização de diferentes técnicas como, por exemplo, *combined profiling* (BERUBE; AMARAL, 2012). Desta maneira os experimentos podem ser mais bem avaliados do ponto de vista estatístico.

Para que o processo de treinamento seja realizado, o *agente externo* deve ser acoplado à JVM no modo `training`. Neste modo, o *bytecode* de cada método do programa é instrumentado com instruções para calcular e registrar o tempo de execução do método. Os dados dos tempos de execução dos métodos são armazenados em um arquivo ao final da execução do programa. Existem dois tipos de procedimentos para a coleta dos tempos de execução: sem amostragem e com amostragem. O procedimento com amostragem é apresentado na seção 4.3.4.

No procedimento sem amostragem, o *agente externo* instrumenta o *bytecode* de cada método do programa com instruções para se obter o tempo de execução do método. Instruções são adicionadas antes de todas as outras do método, para capturar o instante de tempo em que o método inicia a sua execução. Outras instruções são adicionadas após todas as outras do método, para capturar o instante de tempo em que o método terminou a execução e para registrar a diferença dos instantes de tempo, obtendo assim o tempo de execução do método. A cada chamada do método é gerado um registro do seu tempo de execução.

Nesta forma de medição, o tempo de execução do método inclui o tempo de execução de outros métodos que são chamados dentro dele. Entretanto, é possível incorporar ao arcabouço novas formas para a medição de tempo de um método como, por exemplo, considerar o tempo do método excluído o tempo de execução dos métodos que são chamados por ele. Para isto, é necessário identificar todos os métodos que são chamados, coletar o tempo de execução de cada um deles e retirar a soma dos tempos coletados do tempo final do método que está sendo analisado.

É possível que alguns métodos sejam chamados muitas vezes, gerando um grande

número de registros de tempo de execução. Por exemplo, o método *getCode()* da classe *Decompressor* do programa *compress* do *benchmark* SpecJVM2008 (SHIV et al., 2009), pode gerar 8.036.342 registros. Armazenar um número excessivo de registros para cada método aumenta o consumo de memória e também gera uma grande quantidade de dados para serem gravados no arquivo de saída. Para diminuir a quantidade de dados armazenados, é possível definir, pelo parâmetro de configuração `sample_size` do agente, a quantidade máxima de registros de tempo de execução que devem ser armazenados para cada método. Quando a quantidade armazenada atingir o valor do `sample_size`, os demais registros são descartados.

A execução do processo de treinamento é feita pelo *script* `training`. Ele recebe como argumento o identificador de um *benchmark* pré-configurado no arcabouço e também o identificador do plano de otimização que deve ser usado para treinamento. Opcionalmente, pode-se informar qual programa do *benchmark* deve ser executado. Se nenhum programa for informado, todos os programas definidos para o *benchmark* serão executados. Como saída, é gerado um arquivo, para cada programa executado, contendo os dados de treinamento. O Algoritmo 1 apresenta o algoritmo utilizado pelo *script* `training`.

```
   Entrada: Benchmark, Nível de Otimização, Programa (opcional)
   Saída: Dados de treinamento dos programas
 1 início
 2 |   para cada programa do benchmark faça
 3 |   |   extraí as características dos métodos
 4 |   |   para cada conjunto de otimizações faça
 5 |   |   |   executa o programa em modo training
 6 |   |   |   coleta o tempo de compilação e o tempo de execução dos métodos
 7 |   |   fim
 8 |   |   processa os dados coletados e gera os dados de treinamento
 9 |   fim
10 fim
```

Algoritmo 1: Algoritmo do *script* `training`.

Embora o processo de treinamento descrito nessa seção tenha como objetivo tentar reduzir o tempo de execução do programa, é possível adaptar o arcabouço para que ele atinja outro objetivos como, por exemplo, a redução do consumo de energia (PALLISTER; HOLLIS; BENNETT, 2015).

4.3.4 Sampler

A coleta de tempos de execução dos métodos utilizando o procedimento sem amostragem, descrito na seção anterior, afeta o desempenho do programa. Para que o desempenho não seja muito afetado e ainda seja possível coletar um número significante de tempos de execução de cada método, pode-se utilizar o procedimento com amostragem. Nesse procedimento, o agente cria uma cópia do método na própria classe e instrumenta o seu *bytecode* para calcular o seu tempo de execução (ARNOLD; RYDER, 2001) . O método original é instrumentado com um desvio condicional para decidir se chama a cópia que está instrumentada ou se continua a sua execução, sem calcular o seu tempo de execução.

O valor utilizado no desvio condicional, quando o método original é chamado, depende do estado de um recurso denominado `Sampler`. Esse recurso altera o seu estado de tempos em tempos e serve como indicativo para que quando um método é chamado, ele decida se deve ou não efetuar uma chamada a sua cópia. O `Sampler` é inicializado se o parâmetro de configuração `sampling` do *agente externo* for *verdadeiro*.

A Figura 10 apresenta o diagrama de estados do `Sampler`. Quando inicializado, ele entra no estado *Com Amostragem*, que indica aos métodos que devem delegar a sua execução para a cópia criada. Após um tempo, definido pelo parâmetro de configuração `w_sampling` do agente, o `Sampler` altera o seu estado para *Sem Amostragem*, que indica aos métodos que devem continuar a sua execução, sem chamar a cópia. Após outro tempo, definido pelo parâmetro de configuração `wo_sampling` do agente, o `Sampler` altera novamente seu estado para *Com Amostragem*.

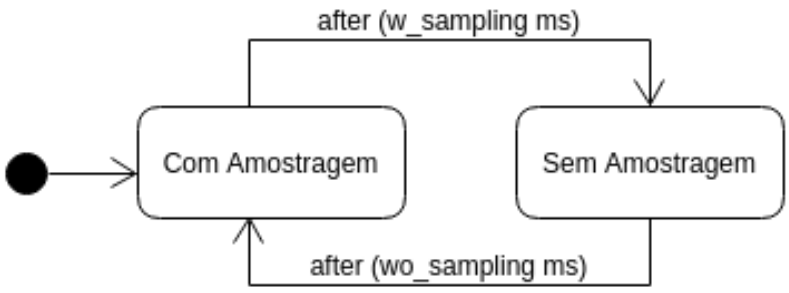

Figura 10: Diagrama de estados do `Sampler`.

O recurso `Sampler` pode ser utilizado para a redução do tempo do processo de treinamento e também durante o processo de aprendizado dinâmico, descrito na seção 4.6.

4.3.5 Machine Learning

O arcabouço permite a utilização de diferentes tipos de algoritmos de aprendizado de máquina. É necessário, portanto, selecionar o algoritmo desejado e especificar os seus parâmetros de configuração. Os algoritmos disponíveis no `ML4JIT` são descritos na seção 5.2.

Após a coleta dos dados de treinamento tem-se, para cada método, um vetor de características $\overrightarrow{F}$ e um vetor $\overrightarrow{O}$ que corresponde aos melhores conjuntos de otimizações para aquele método. Dado o vetor de características de um novo método, utiliza-se o recurso `Machine Learning` para se obter o conjunto de otimizações que serão aplicadas ao método. A utilização desse recurso segue três etapas: (*i*) seleção do algoritmo de aprendizado de máquina; (*ii*) treinamento do modelo; (*iii*) integração do modelo a uma JVM.

Quando selecionado um tipo de algoritmo de aprendizado de máquina, ele deve ser treinado. Os dados para o treinamento são coletados através recurso `training`, descrito na seção 4.3.3. Esses dados devem ser informados para o *script* `ml`, responsável por treinar o modelo de aprendizado.

Após o treinamento, o algoritmo de aprendizado de máquina pode ser utilizado para selecionar, para cada método do programa que se deseja avaliar e testar, o conjunto de otimizações a ser aplicado. Esse processo de seleção pode ser feito tanto de forma dinâmica (*online*) quando de forma estática (*offline*).

Na forma estática, é gerado um arquivo contendo o conjunto de otimizações que deve ser aplicado a cada método do programa. Esse arquivo deve então informado à JVM para que no momento da compilação do método sejam aplicadas as otimizações desejadas. O processo de geração desse arquivo é feito pelo *script* `ml_outputs`. Nesta forma, não é necessário o uso do agente externo do arcabouço.

Na forma dinâmica, a JVM solicita ao recurso `Machine Learning` do arcabouço para obter o conjunto de otimizações para o método que será compilado. A solicitação é feita a um servidor, através de uma conexão *socket*. A JVM deve conectar-se ao servidor e informar o vetor de características do método. O servidor retorna o conjunto de otimizações a ser aplicado para aquele método. Para auxiliar nesse processo, o agente externo deve ser acoplado à JVM.

Tabela 3: Parâmetros de configuração do agente.

Nome	**Tipo**	**Valor Padrão**	**Descrição**
debug	lógico	false	Habilita o registro de log com detalhes sobre a operação do agente.
record_code	lógico	false	Grava no arquivo texto *code.txt* o *bytecode* dos métodos após serem instrumentalizados.
sampling	lógico	false	Habilita o modo de amostragem.
mode	texto	counter	Modo em que o agente será utilizado. Os valores possíveis são: *counter*, *training* ou *ml*.
skip	texto	*null*	Arquivo texto contendo classes e métodos que não devem ser analisados pelo agente.
path	texto	.	Caminho do diretório onde serão gravados os arquivos gerados pelo agente.
features	texto	./features.csv	Arquivo contendo as características dos métodos extraídas de forma estática.
classpath	texto	*null*	Lista de caminhos ou arquivos `.jar` de *classpath* do programa que será rodado. Os dados da lista devem ser separados por dois pontos.
sample_size	inteiro	100	Tamanho máximo de amostras que devem ser registradas para um método.
w_sampling	inteiro	10	Quantos milissegundos roda com amostragem.
wo_sampling	inteiro	90	Quantos milissegundos roda sem amostragem.

4.3.6 Configurações do Agente

O agente externo possui um conjunto de parâmetros de configuração que são utilizados para definir o seu comportamento. Todos os parâmetros são opcionais e caso o valor do parâmetro não seja informado, assume-se um valor padrão.

Os argumentos são informados de forma textual, via linha de comando, nas opções do comando que acopla o agente a JVM. O formato <*nome*>=<*valor*> deve ser usado para a definição de cada argumento. Caso haja mais de um argumento, eles devem ser separados por ponto-e-vírgula. A ordem dos argumentos não é considerada. A Tabela 3 apresenta o conjunto de parâmetros de configuração disponíveis.

4.4 Integração com a Máquina Virtual Java

O modelo de aprendizado de máquina utilizado na etapa de avaliação e testes do programa deve ser integrado à máquina virtual Java. Diferentes estratégias podem ser adotadas para a realização dessa integração. Em (CAVAZOS; O'BOYLE, 2006), as heurísticas geradas pelo processo de aprendizado são instaladas diretamente no compilador. Em

(SANCHEZ et al., 2011), o modelo de aprendizado de máquina é executado em um processo separado e uma comunicação entre ele e a JVM é feita utilizando pipe nomeado (*named pipes*). No `ML4JIT`, pode-se escolher entre duas estratégias adotadas: (i) instalação dos dados de aprendizado diretamente na JVM; (ii) utilização de um servidor para capturar os dados de aprendizado.

Na primeira estratégia, os dados de aprendizado são instalados na JVM informando a ela um arquivo contendo o conjunto de otimizações que deve ser aplicado a cada um dos métodos do programa. Para isso, a máquina virtual deve possuir a funcionalidade *JVM04*, descrita na seção 4.5. O arquivo que deve ser informado à JVM é gerado pelo *script* `ml_outputs`, conforme descrito na seção 4.3.5.

No momento em que a máquina virtual decide compilar um método, é verificado nos dados informados, qual o conjunto de otimizações que deve ser aplicado. O compilador prepara então um plano de compilação habilitando somente as otimizações desejadas e realiza a compilação do método.

Na segunda estratégia, os dados de aprendizado são obtidos por meio de uma comunicação com um servidor, que pode rodar na mesma máquina ou em uma máquina separada. O servidor contém o modelo de aprendizado e é capaz de retornar um valor que indica qual conjunto de otimizações deve ser aplicado a um método, a partir de suas características.

Quando a JVM decide compilar um método, ela deve conectar-se ao servidor e informar o vetor de características do método. O servidor retorna um valor que indica o conjunto de otimizações que deve ser aplicado àquele método. Para auxiliar nesse processo, o agente externo deve ser acoplado à JVM.

4.5 Modificações na Máquina Virtual Java

O funcionamento adequado do arcabouço `ML4JIT` depende de alguns dados e funcionalidades que são de responsabilidade da máquina virtual Java. A seguir, são apresentadas as principais funcionalidades que a JVM deve possuir para que possam ser utilizada com o arcabouço.

JVM01 - Registrar, em um arquivo, o tempo de compilação de cada método do programa. O tempo de compilação dos métodos pode ser usado na etapa de treinamento dos algoritmos de aprendizado de máquina e usado também nos resultados dos experimentos, para verificar se com o uso do conjunto de otimizações sugerido pelo arcabouço reduziu o

tempo de compilação do programa.

JVM02 - Registrar, em um arquivo, o tempo de execução do programa, excluindo o tempo gasto nos procedimentos de inicialização da JVM. Esse tempo de execução deve começar a contar a partir do momento em que a máquina virtual chama o método `main` do programa até o final da execução do programa. Ele é utilizado para verificar se com o uso do conjunto de otimizações sugerido pelo arcabouço reduziu o tempo de execução do programa.

JVM03 - Criar um plano de otimização específico para cada método do programa. A JVM deve ser capaz de, a partir dos dados informados pelo recurso de aprendizado de máquina, aplicar um plano de otimização diferente para cada método do programa.

JVM04 - Carregar os dados gerados pelo recurso de aprendizado de máquina, contendo o conjunto de otimizações específico para cada método do programa. Quando o recurso `Machine Learning` do arcabouço é usado de forma estática, ele gera um arquivo contendo uma tabela que mapeia cada método com o conjunto de otimizações que deve ser aplicado a ele. A JVM deve ser capaz de ler esse arquivo e utilizar os dados para criar o plano de otimização específico para cada método.

JVM05 - Operar em diferentes modos de execução: padrão, treinamento, aprendizado de máquina, aprendizado dinâmico. Cada recurso contido no arcabouço necessita que a JVM opere de forma diferente. Por exemplo, quando em treinamento, a JVM deve aplicar o mesmo plano de otimização para todos os métodos do programa e, quando o recurso de `Machine Learning` estiver sendo usado, a máquina virtual deve aplicar um plano de otimização específico para cada método do programa.

As modificações sugeridas para a máquina virtual Java foram implementadas na Jikes RVM, conforme apresentado na seção 5.3.

4.6 Aprendizado Dinâmico

A estrutura do arcabouço apresentada até o momento utiliza técnicas de aprendizado de máquina supervisionado. Com o uso destas técnicas o aprendizado é feito através de exemplos fornecidos por algum supervisor externo em uma fase de treinamento (MITCHELL, 1997). No caso do uso de aprendizado supervisionado em compiladores, o treinamento é feito através de uma busca aleatória na tentativa de se obter os melhores conjuntos de otimizações para um programa ou parte dele. Esse tipo de busca não cobre todo o espaço de otimizações e ainda é necessário selecionar o mesmo conjunto de oti-

mizações muitas vezes para verificar se ele realmente melhora o desempenho do programa (OGILVIE et al., 2017). Além disso, o tempo gasto na fase de treinamento é grande, e caso haja alguma mudança na plataforma de execução, o treinamento do modelo tem de ser refeito.

Para lidar com tais questões, o arcabouço possui uma estrutura para permitir que pesquisas experimentais sejam realizadas em um processo de aprendizado dinâmico; isto é, ao invés de se obter os dados para treinamento do modelo de aprendizado de forma estática, executando uma busca exaustiva ou aleatória sobre o espaço de otimizações, esses dados poderiam ser obtidos em tempo de execução do programa.

Durante a execução do programa, diferentes conjuntos de otimizações podem ser testados para um mesmo método do programa. As informações coletadas com a aplicação desses diferentes conjuntos de otimizações são usadas para alimentar modelos de aprendizado de máquina que permitem aprendizado dinâmico como, por exemplo, aprendizado ativo (SETTLES, 2012; OGILVIE et al., 2017) ou aprendizado por reforço (SUTTON; BARTO, 2017). A utilização de aprendizado dinâmico pode permitir que o tempo gasto para o treinamento do modelo seja reduzido e também que o modelo de aprendizado consiga adaptar-se mais rapidamente a mudanças na plataforma de execução.

Para que diferentes conjuntos de otimizações para um método sejam testados durante a execução de um programa, a máquina virtual deve permitir ao arcabouço que ele indique que um determinado método deve ser recompilado e também qual conjunto de otimizações deve ser aplicado nessa recompilação. A JVM deve também capturar o tempo de compilação e de execução do método após a recompilação. A proposta de arquitetura para aprendizado dinâmico no `ML4JIT` é apresentada na seção 5.4.

O uso de aprendizado dinâmico durante a execução do programa pode afetar muito o seu desempenho, uma vez que é necessário instrumentar todos os métodos do programa para coletar dados do seu perfil de execução. Para que o desempenho não seja muito afetado, o recurso `Sampler` do arcabouço tem como objetivo principal chamar a versão instrumentada do método somente algumas vezes durante a execução do programa, reduzindo o impacto em tempo execução.

Uma forma complementar para diminuir o impacto do uso de aprendizado dinâmico é a de instrumentar somente os métodos que possam se beneficiar de forma prática das otimizações aplicadas pelo compilador e também os métodos nos quais a adição de instruções não causem muita interferência na medição do tempo de execução do método. O `ML4JIT` possui um recurso, apresentado na próxima seção, que permite identificar auto-

maticamente métodos que possam ser excluídos do processo de instrumentação de código.

4.7 Nano-patterns

A instrumentação de código nos métodos do programa adiciona uma sobrecarga de tempo a esses métodos, interferindo no seu tempo de execução e provocando ruído em uma medição mais precisa do tempo de execução real do método. Em alguns casos, os códigos adicionados ao método têm um tempo de execução maior que o das instruções originais. Ou, o tempo de execução do método é muito baixo, por exemplo, menor que 1ms, não sendo satisfatório incluir instruções adicionais para a medição de tempo.

Tabela 4: Lista de *nano-patterns* apresentada em (SINGER et al., 2010). Os nomes em negrito foram adicionados em (MIGNON; ROCHA, 2017).

Nome	**Descrição**
NoParams	não tem argumentos
NoReturn	returna void
Recursive	chama a si mesmo recursivamente
SameName	chama outro método com o mesmo nome
Leaf	não chama nenhum método
ObjectCreator	cria novos objetos
FieldReader	lê valores de campos de um objeto
FieldWriter	escreve valores em campos de um objeto
TypeManipulator	usa operações de conversão de tipo ou *instanceof*
StraightLine	sem desvios no corpo do método
Looping	um ou mais laços no corpo do método
Exceptions	pode lançar uma exceção não tratada
LocalReader	lê valores de uma variável local
LocalWriter	escreve valores em uma variável local
ArrayCreator	cria um novo vetor
ArrayReader	lê valores de um vetor
ArrayWriter	escreve valores em um vetor
Thread	método `run()` de uma classe que estende java.lang.Thread
Arithmetic	tem operação aritmética

Uma técnica para minimizar o problema é a de instrumentar somente os métodos que possam se beneficiar de forma prática das otimizações aplicadas pelo compilador e também os métodos nos quais a adição de instruções não causem muita interferência na medição do tempo de execução do método. O `ML4JIT` possui um recurso, baseado em nano-patterns (SINGER et al., 2010), que permite identificar automaticamente métodos que possam ser excluídos do processo de instrumentação de código (MIGNON; ROCHA, 2017). Eles são descritos pela composição de *nano-patterns* ou simplesmente pelo valor verdadeiro de um único *nano-patterns*.

Tabela 5: Lista de padrões para a caracterização de métodos.

Nome	**Descrição**
ACCESS	Métodos de acesso a dados de atributos de um objeto.
MODIFY	Métodos de modificação de dados de atributos de um objeto.
SAME NAME	Métodos que chamam outro método com o mesmo nome.
RECURSIVE	Métodos recursivos
THREAD	Métodos run() de uma classe que estende a classe Thread

Nano-patterns são propriedades de métodos Java que são (SINGER et al., 2010):

- *simples*: elas podem ser identificadas pela inspeção manual de um desenvolvedor Java ou podem ser extraídas de forma trivial através de uma ferramenta de análise;
- *estáticas*: elas devem ser determinadas pela análise do bytecode de um programa Java, sem qualquer informação de contexto de execução do programa;
- *binárias*: cada propriedade tem somente o valor verdadeiro ou falso para um dado método;

Essas propriedades são ditas *rastreáveis*, isto é, elas podem ser expressas como uma condição formal simples sobre os atributos, tipos, nome e corpo de um método Java (GIL; MAMAN, 2005). Elas também podem ser identificadas de forma automática através de uma análise estática do *bytecode* Java. A Tabela 4 apresenta a lista dos *nano-patterns* fundamentais. Os primeiros 17 *nano-patterns* são apresentados em (SINGER et al., 2010). Os dois últimos, apresentados em negrito, são padrões identificados em (MIGNON; ROCHA, 2017).

A combinação lógica dos *nano-patterns* permite construir padrões mais complexos denominados *composite nano-patterns*. Por exemplo, o *PureMethod composite nano-pattern*, extraído de (SINGER et al., 2010), pode ser especificado da seguinte forma:

$$\neg \text{ FieldWriter} \wedge \neg \text{ ArrayWriter} \wedge \neg \text{ ObjectCreator} \wedge \neg \text{ ArrayCreator} \wedge \text{Leaf}$$

A Tabela 5 apresenta a lista de padrões para a caracterização de métodos que podem ser suprimidos do processo de instrumentação de código. Os padrões foram identificados em (MIGNON; ROCHA, 2017) e são utilizados no `ML4JIT`. Esta lista não pretende ser exaustiva, novos padrões podem ser identificados e facilmente incorporados ao arcabouço. A justificativa para a utilização de cada padrão é apresentada a seguir.

O padrão *ACCESS* tem por objetivo identificar métodos que acessam atributos do objeto e retornam o seu valor. Além disso, pode identificar que o método faz uma chamada a outro método para obter um valor e retorná-lo. Caso o método tenha laço ou operação aritmética, ele deve ser instrumentado. A definição desse tipo de método é definida conforme o *composite nano-pattern* a seguir.

$$\neg \text{ NoReturn} \wedge \text{FieldReader} \wedge\neg \text{ Looping} \wedge \text{LocalReader} \wedge\neg \text{ Arithmetic}$$

O padrão *MODIFY* tem por objetivo identificar métodos que modificam os atributos do objeto. Esse tipo de método não deve ter nenhum valor de retorno e não efetua chamada a nenhum outro método. Caso o método tenha laço ou operação aritmética, ele deve ser instrumentado. A definição desse tipo de método é definida conforme o *composite nano-pattern* a seguir.

$$\text{NoReturn} \wedge \text{Leaf} \wedge \text{FieldWriter} \wedge\neg \text{ Looping} \wedge \text{LocalReader} \wedge\neg \text{ Arithmetic}$$

O padrão *SAME NAME* tem por objetivo identificar os métodos que chamam outro método com o mesmo nome. Em geral, esse tipo de método não tem nenhum tipo de operação relevante, delegando para outro método de mesmo nome a execução das operações necessárias. Esse tipo de método é identificado se o *nano-pattern SameName* for verdadeiro.

O padrão *RECURSIVE* tem por objetivo identificar métodos que são auto-recursivos. Devido a forma de medição de tempo de execução de método do `ML4JIT` , a instrumentação deste tipo de método faria com que todas as chamadas recursivas tivessem seu tempo medido, dificultando assim a acurácia da medição do tempo total de execução do método. Este padrão não identifica recursões que ocorrem de forma indireta. Para isto, seria necessário analisar o grafo das chamadas dos métodos. Métodos auto-recursivos são identificados se o *nano-pattern Recursive* for verdadeiro.

O padrão *THREAD* tem por objetivo identificar métodos *run()* de classes que estendem a classe *Thread*. Em geral, esse tipo de método executa infinitamente até que é parado abruptamente ou, delega a operação para outros métodos. Esse tipo de método é identificado se o *nano-pattern Thread* for verdadeiro.

4.8 Conclusão

Este capítulo apresentou uma visão geral da estrutura e dos recursos principais presentes no arcabouço `ML4JIT`. A ideia do arcabouço é possibilitar a construção de um laboratório para sejam realizadas pesquisas com aprendizado de máquina em compiladores JIT para a linguagem Java com a intenção de se descobrir o melhor conjunto de otimizações que pode ser aplicado de forma específica a cada método de um programa. O próximo capítulo apresenta os detalhes de projeto e implementação do arcabouço.

5 PROJETO E IMPLEMENTAÇÃO DO ARCABOUÇO

Este capítulo apresenta detalhes do projeto e da implementação dos componentes do arcabouço `ML4JIT`[1], apresentados no capítulo anterior.

5.1 Arquitetura do Agente Externo

O agente externo presente no `ML4JIT` foi desenvolvido utilizando a API Java Agent (ORACLE, 2016a). Optou-se pelo uso dessa API por ela permitir que o agente seja escrito totalmente em Java. Como alternativa, poderia ser utilizada a JVMTI (*JVM Tool Interface*) (ORACLE, 2016b), entretanto, uma parte do agente deveria ser escrita em linguagem C/C++.

As classes que implementam o agente estão contida no arquivo `ml4jit.jar`. Essas classes são responsáveis por analisar e instrumentar o *bytecode* dos métodos do programa. Para que o agente seja executado, ele deve ser acoplado à JVM no momento da execução do programa através da opção `-javaagent`. A Figura 11 apresenta a dinâmica operacional do agente.

No momento em que o programa Java é carregado, a máquina virtual (JVM) aciona o método `premain` do `Agent`. Este método é o ponto de entrada para o agente e é responsável por informar a JVM qual classe de transformação (implementação da interface `ClassFileTransformer`) deve ser utilizada. Durante a execução do programa, as classes são carregadas, uma única vez, no momento em que é necessária a sua utilização. O controle de carregamento das classes é feito pelo `ClassLoader` da JVM. No momento em que uma nova classe é carregada, o `Classloader` aciona o `Agent` solicitando que ele realize o processo de transformação da classe e de seus métodos. O `Agent` então aciona uma de suas classes de transformação para realizar a instrumentação dos métodos da classe. Após

[1] Os arquivos apresentados neste capítulo estão disponíveis em: https://github.com/amignon/ml4jit.git.

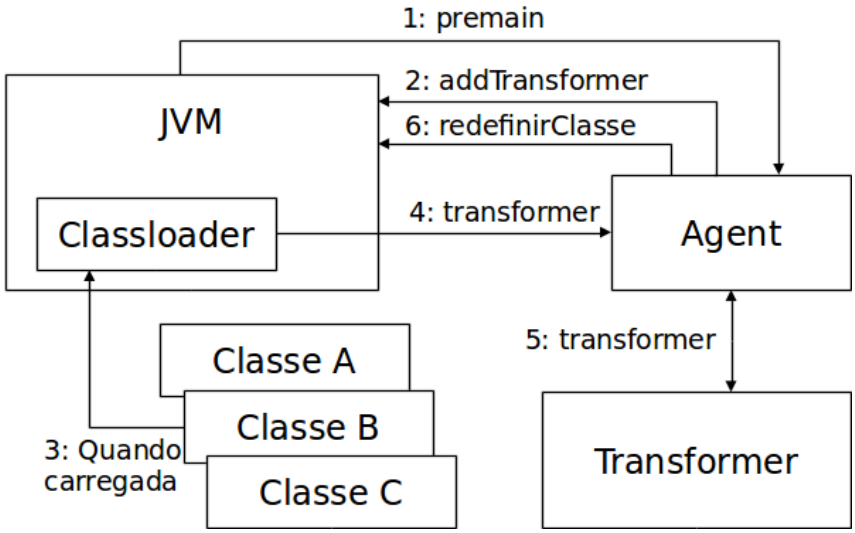

Figura 11: Dinâmica operacional do agente.

a instrumentalização o `Agent` envia a classe transformada para a JVM.

A Figura 12 apresenta uma visão geral da arquitetura do agente. A classe `Agent` implementa o método `premain`, necessário para criar um *Java Agent*. A classe de transformação a ser utilizada é criada pela classe `TransformerFactory`, dependendo do modo de uso do agente, informado nos parâmetros de configuração, descritos na seção 4.3.6. Ela cria uma instância de uma classe que estende a classe abstrata `Transformer`. Três classes estendem a `Transformer`: `CounterTransformer`, `TrainingSamplingTransformer` e `TrainingTransformer`.

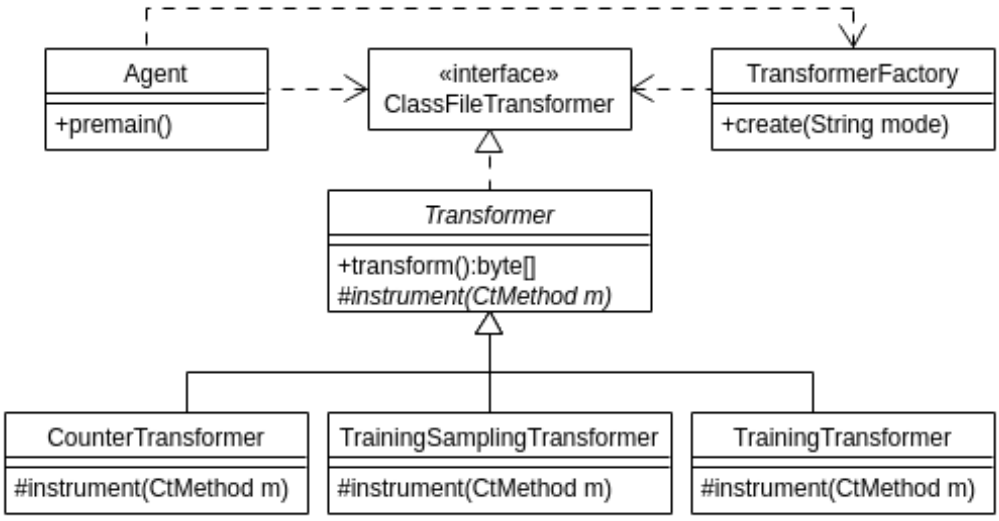

Figura 12: Visão geral da arquitetura do agente.

A classe abstrata `Transformer` implementa a operação `transform` contida na interface `ClassFileTransformer` como um *Template Method* (GAMMA et al., 1995). As

classes que estendem a `Transformer` devem ser responsáveis por implementar o método `instrument` de acordo com o tipo de instrumentação desejada.

A instrumentação do *bytecode* dos métodos é feita utilizando o arcabouço Javassist (CHIBA, 2000). Ele permite a análise e edição de *bytecodes* em Java. Decidiu-se utilizar esse arcabouço por sua facilidade de uso e porque as instruções de instrumentalização podem ser escritas em Java. Entretanto, o ML4JIT pode ser adaptado para a utilização outros arcabouços de instrumentação de *bytecode* como, por exemplo, ASM (KULESHOV, 2007) e BCEL (DAHM, 2001).

A arquitetura apresentada permite que novas classes de transformação sejam adicionadas facilmente ao arcabouço. Para isso, é necessário criar uma classe que estende a classe abstrata `Transformer` e implementar o método abstrato `instrument`. É preciso também alterar o método `create` da classe `TransformerFactory` para que ele possa criar uma instância da classe adicionada ao arcabouço.

As próximas seções apresentam os detalhes de projeto e implementação dos recursos disponíveis no arcabouço, descritos no capítulo 4.

5.1.1 Counter

O recurso `Counter` contido no `ml4jit.jar` é acionado quando o agente é utilizado no modo `counter`. Nesse modo, cada método do programa é instrumentalizado com o objetivo de contar a quantidade de vezes em que ele é chamado durante a execução do programa. A Figura 13 apresenta o diagrama de classes do recurso `Counter`.

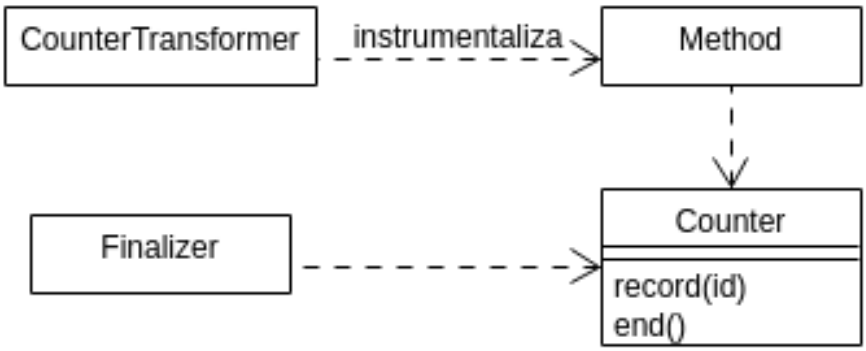

Figura 13: Diagrama de classes do recurso `Counter`.

A classe `CounterTransformer` instrumentaliza o *bytecode* de cada método do programa com uma instrução, após todas as outras, para registrar que ele foi chamado. A Figura 14 apresenta o código do método `transform` da classe `CounterTransformer`.

Quando o método é chamado durante a execução do programa, ele aciona o método estático `record` da classe `Counter`. Como argumento, é informado o identificador [2] do método. A classe `Counter` contém um mapeamento entre o identificador do método e a quantidade de vezes que ele é chamado.

```
protected void transform(CtMethod m) throws Exception {
  int id = Util.getId(m);
  Counter.addId(id, Util.getNome(m));
  m.insertBefore("ml4jit.counter.Counter.record(" + id + ");");
}
```

Figura 14: Código do método `transform` da classe `CounterTransformer`.

Ao final da execução do programa a classe `Finalizer` aciona o método `end()` da classe `Counter`. Esse método gera um arquivo texto (*counter.txt*) com os dados obtidos durante a execução. O cabeçalho do arquivo contém a quantidade de métodos analisados e a quantidade total de chamadas de métodos. Cada registro do arquivo representa um método e contém: o *id*, a assinatura, a quantidade de vezes que o método foi chamado.

5.1.2 Training

O recurso `Training` contido no `ml4jit.jar` é acionado quando o agente é utilizado no modo `training`. Nesse modo, os métodos do programa são instrumentalizados para obter o seu tempo de execução. A Figura 15 apresenta o diagrama de classes do recurso `Training`.

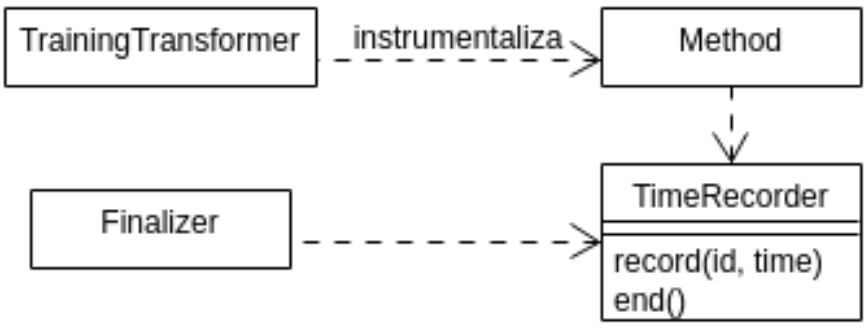

Figura 15: Diagrama de classes do recurso `Training`.

A classe `TrainingTransformer` instrumentaliza o *bytecode* de cada método do programa com instruções para obter o tempo de execução do método. A Figura 16 apresenta

[2] O identificador do método é gerado a partir do *hashcode* da `String` que representa a assinatura do método.

```
protected void transform(CtMethod m) throws Exception {
  m.addLocalVariable("___ini___", CtClass.longType);
  m.addLocalVariable("___dif___", CtClass.longType);
  m.insertBefore("___ini___ = System.nanoTime();");
  int id = Util.getId(m);
  StringBuilder sb = new StringBuilder();
  sb.append("{___dif___ = System.nanoTime() - ___ini___;");
  sb.append("ml4jit.training.TimeRecorder.record("+ id +", ___dif___);}");
  m.insertAfter(sb.toString());
}
```

Figura 16: Código do método `transform` da classe `TrainingTransformer`.

o código do método `transform` da classe `TrainingTransformer`. Quando o método é chamado durante a execução do programa, ele aciona o método estático `record` da classe `TimeRecorder`. Como argumento, é informado o identificador do método e o seu tempo de execução. A classe `TimeRecorder` contém um mapeamento entre o identificador do método e os tempos de execução obtidos.

Alternativamente, a implementação do método `transform`, apresentado na Figura 16, pode ser modificada para verificar se o código do método instrumentado é executado em diferentes *Threads*. Se isso ocorrer, a medição do tempo de execução do método fica prejudicada e o melhor a ser feito é descartar o registro dessa medição. Entretanto, os programas utilizados para a validação do arcabouço não apresentaram esse tipo de comportamento.

Ao final da execução do programa a classe `Finalizer` aciona o método `end()` da classe `TimeRecorder`. Esse método gera um arquivo texto (*time.txt*) com os dados obtidos durante a execução. Cada registro do arquivo contém: o *id* do método e o tempo de execução (em nanosegundos) de uma chamada ao método.

5.1.3 Sampler

O recurso `Sampler` contido no `ml4jit.jar` é inicializado quando o parâmetro de configuração `sampling` do agente for *verdadeiro*. Esse recurso pode ser utilizado quando o agente é executado no modo `training`. Ele permite que a coleta dos tempos de execução dos métodos do programa seja por amostragem, conforme descrito na seção 4.3.4. A Figura 17 apresenta o diagrama de classes do recurso `Sampler`.

A classe `TrainingSamplingTransformer` é responsável por:

1. inicializar a classe `Sampler`, que de tempos em tempos modifica o seu estado. A

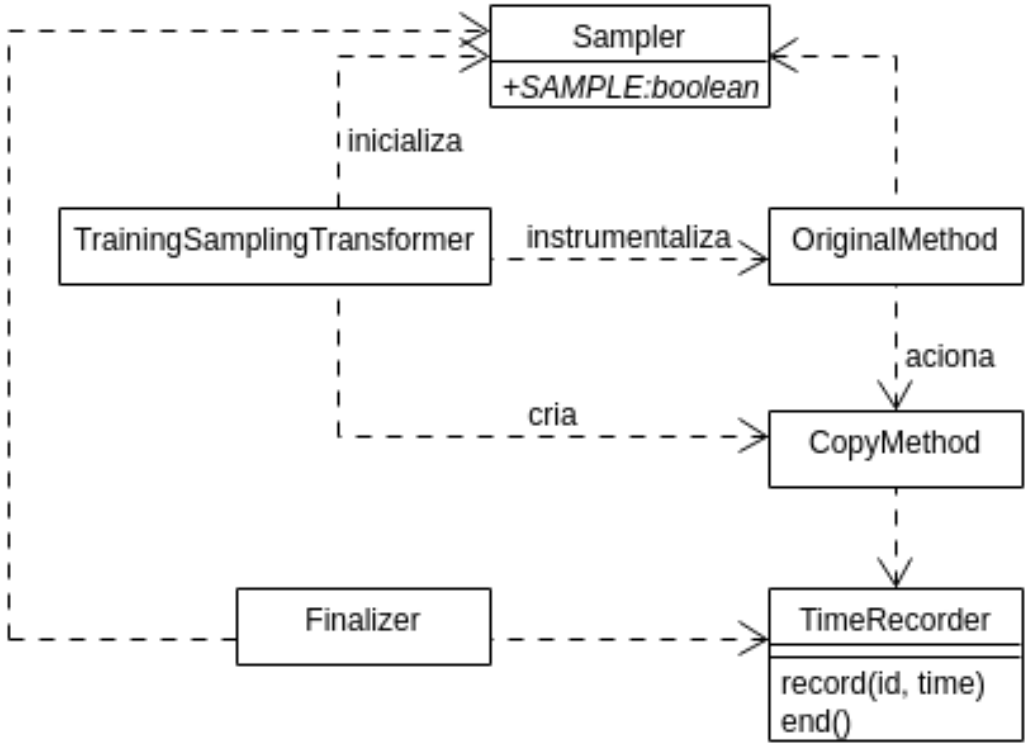

Figura 17: Diagrama de classes do recurso `Sampler`.

descrição dos estados do `Sampler` é apresentada na seção 4.3.4.

2. criar uma cópia de cada método do programa com o objetivo de instrumentar seu *bytecode* para se obter o tempo de execução do método.

3. instrumentar o método original para que quando for acionado ele decida se deve chamar o método copiado, que está instrumentalizado, ou se deve continuar a executar o seu código, que não está instrumentalizado. Essa decisão é baseada no estado do atributo estático `SAMPLE` da classe `Sampler`.

O código do método `transform` da classe `TrainingSamplingTransformer` é apresentado na Figura 18. Quando o método copiado é chamado durante a execução do programa, ele aciona o método estático `record` da classe `TimeRecorder`. Como argumento, é informado o identificador do método *original* e o seu tempo de execução. A classe `TimeRecorder` contém um mapeamento entre o identificador do método e os tempos de execução obtidos.

Ao final da execução do programa a classe `Finalizer` aciona o método `end()` da classe `TimeRecorder`. Esse método gera um arquivo texto (*time.txt*) com os dados obtidos durante a execução. Cada registro do arquivo contém: o *id* do método e o tempo de

```
protected void transform(CtMethod m) throws Exception {
  // create a copy of the method
  CtClass c = m.getDeclaringClass();
  CtMethod copy = CtNewMethod.copy(m, m.getName() + "_COPY", c, null);
  c.addMethod(copy);
  // instrument the original method
  StringBuilder sb = new StringBuilder();
  sb.append("{if(ml4jit.Sampler.SAMPLE) {");
  if (!m.getReturnType().getName().equals("void")) {
    sb.append("return ");
  }
  sb.append(m.getName() + "_COPY();");
  if (m.getReturnType().getName().equals("void")) {
    sb.append("return;");
  }
  sb.append("}}");
  m.insertBefore(sb.toString());
  // instrument the copy
  copy.addLocalVariable("___ini___", CtClass.longType);
  copy.addLocalVariable("___dif___", CtClass.longType);
  copy.insertBefore("___ini___ = System.nanoTime();");
  int id = Util.getId(m);
  StringBuilder sb = new StringBuilder();
  sb.append("{___dif___ = System.nanoTime() - ___ini___;");
  sb.append("ml4jit.training.TimeRecorder.record("+ id +", ___dif___);}");
  copy.insertAfter(sb.toString());
}
```

Figura 18: Código do método `transform` da classe `TrainingSamplingTransformer`.

execução (em nanosegundos) de uma chamada ao método. A classe `Finalizer` também é responsável por finalizar a execução do `Sampler`.

5.1.4 Features Extractor

O recurso `Features Extractor` contido no `ml4jit.jar` é responsável por extrair as características dos métodos para serem usadas nos algoritmos de aprendizado de máquina. A descrição das características extraídas de cada método são apresentadas seção 4.3.2. Elas podem ser extraídas tanto de forma estática quanto de forma dinâmica. A Figura 19 apresenta o diagrama de classes desse recurso.

A classe `Code` é responsável por analisar o *bytecode* de um método, em um único passo, e criar um objeto da classe `Features`, que representa as características extraídas do método. Esse objeto é então associado ao um objeto `Mapper`, que relaciona um método com as suas características.

Quando o recurso é utilizado de forma estática, a classe `FeaturesExtractor` deve ser acionada como um programa Java. Ela deve receber os caminhos onde os arquivos do

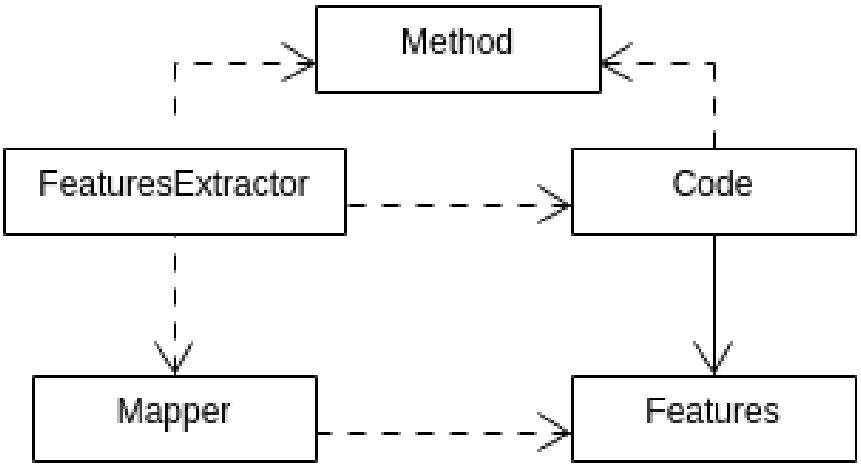

Figura 19: Diagrama de classes do recurso `Features Extractror`.

programa estão armazenados, através da opção `-cp` da máquina virtual Java. Os arquivos são então analisados e, para cada método do programa, um vetor de características é gerado. Ao final do programa, os dados são armazenados em um arquivo texto *features.csv*.

A utilização do recurso de forma dinâmica, isto é, quando se deseja extrair as características dos métodos em tempo de execução do programa, deve ser feita por uma instância de uma classe que estende a classe abstrata `Transformer`. Nela, a classe `Code` de ser acionada para que as características dos métodos sejam extraídas e depois sejam armazenadas em um objeto `Mapper`. A extração de características de forma dinâmica pode ser utilizada quando se deseja realizar pesquisas experimentais com aprendizado dinâmico, descrito na seção 4.6.

5.2 Arquitetura dos Scripts

O `ML4JIT` é composto por um conjunto de *scripts* e módulos, desenvolvidos em linguagem de programação Python 2.7. Os *scripts* são responsáveis pelo controle de execução do arcabouço. Os módulos contêm classes e funções auxiliares para o desenvolvimento dos *scripts*.

A Figura 20 apresenta a estrutura geral dos *scripts* e módulos. Criou-se uma arquitetura para facilitar a configuração dos programas que são utilizados nas pesquisas experimentais realizadas com o arcabouço. Ela permite que novos programas sejam facilmente adicionado e configurado. A seguir, apresenta-se uma breve descrição de cada um dos *scripts*

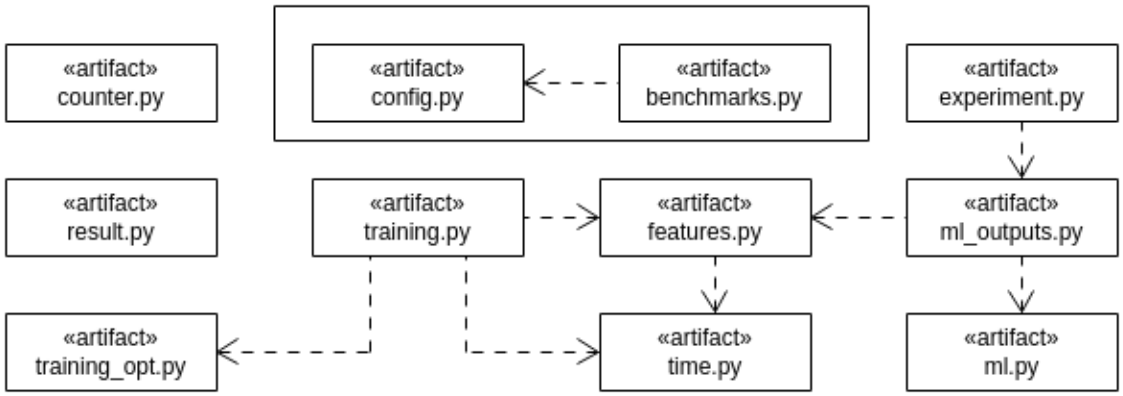

Figura 20: Arquitetura dos *scripts*.

O `config.py` é utilizado como um arquivo de configuração. Os valores atribuídos a cada parâmetro de configuração definem o comportamento do arcabouço e são são acessados por todos os outros *scripts*.

No `ML4JIT`, um programa ou um conjunto de programas é denominado de `Benchmark`. O `benchmarks.py` contém uma estrutura para representar *benchmarks* e programas que são executados pelo arcabouço. Esse artefato também é utilizado por todos os outros *scripts*.

O `counter.py` é responsável por controlar a execução do arcabouço no modo *counter*. Ele recebe o programa que se quer analisar e cria o comando adequado para acoplar o *agente* à JVM e acionar a máquina virtual para a execução do programa.

O `features.py` é responsável por controlar a execução do processo de extração das características dos métodos de um programa de forma estática. Ele recebe o `Benchmark` que se quer analisar e cria o comando adequado para chamar o programa `FeaturesExtractor` contido no arquivo `ml4jit.jar`.

Na fase de treinamento, o `features.py` é responsável por gerar o arquivo que contém os melhores conjuntos de otimizações identificados para cada método do programa analisado, sendo que cada conjunto identificado é associado ao vetor de características do respectivo método. Para auxiliar nesse processo de identificação, ele utiliza o módulo `time.py`, que processa os tempos de execução e de compilação coletados.

O `ml_outputs.py` é responsável por gerar um arquivo contendo o conjunto de otimizações que deve ser aplicado para cada um dos métodos do programa. Esse arquivo é utilizado quando se decide instalar os dados de aprendizado diretamente na máquina virtual. A identificação do conjunto de otimizações que deve ser aplicado a cada método

é feita pelo módulo `ml.py`.

Este módulo cria um modelo de aprendizado de máquina supervisionado a partir do arquivo gerado pelo `features.py` e utiliza esse modelo para predizer qual o conjunto de otimizações deve ser aplicado. Os algoritmos utilizados para o aprendizado de máquina supervisionado estão contidos na biblioteca *scikit-learn* (PEDREGOSA et al., 2011). Diferentes tipos de algoritmos desta biblioteca com distinos parâmetros de configuração podem ser utilizados para as pesquisas experimentais. Atualmente, os algoritmos que podem ser utilizados no arcabouço são: *DecisionTreeClassifier*, *KNeighborsClassifier*, *RadiusNeighborsClassifier*, *RandomForestClassifier* e *SVC*. Entretanto, novos algoritmos podem ser adicionados ao `ML4JIT`.

O `training.py` é responsável por controlar a execução do processo de treinamento. Ele recebe o `Benchmark` e o nível de otimização que se quer utilizar para gerar os dados de treinamento e executa o procedimento descrito no Algoritmo 1 da seção 4.3.3. Esse *script* utiliza o módulo `training_opt.py` para representar o conjunto de otimizações de um determinado nível. Esse módulo habilita ou desabilita as opções de otimizações do nível selecionado.

O `experiment.py` é responsável por controlar a execução dos programas quando se quer testá-los utilizando o conjunto de otimização definido para cada método pelo modelo de aprendizado de máquina supervisionado. Ele recebe o `Benchmark` e o nível de otimização que se quer analisar e coleta os tempos de compilação e de execução de cada um dos programas, armazenando-os em arquivos. Para comparação, cada programa é executado de duas formas diferentes: uma em que todas as otimizações do nível selecionado estão habilitadas e outra em que as otimizações são definidas pelo modelo de aprendizado de máquina. Os dados dos arquivos gerados por esse *script* são utilizados pelo `result.py` para gerar os gráficos comparativos dos resultados dos experimentos.

5.3 Modificações na Jikes RVM

A máquina virtual Jikes RVM foi definida para ser utilizada junto com o arcabouço `ML4JIT`. Escolheu-se essa máquina virtual por ela ser de código aberto e por ser escrita quase que totalmente em linguagem Java. A Jikes RVM possui diversos tipos de otimização de código, agrupados em níveis de otimização, permitindo a realização das pesquisas experimentais propostas neste trabalho. Entretanto, para que a Jikes RVM pudesse ser empregada, foi necessário realizar alterações em seu código. As alterações

foram realizadas na versão 3.1.4.

Algumas classes contidas no código fonte da Jikes RVM foram modificadas e novas classes foram adicionadas para implementar as funcionalidades descritas na seção 4.5. Todas as novas classes estão contidas no pacote `ml4jit.rvm`, adicionado ao código fonte da Jikes RVM. A Figura 21 apresenta o diagrama das classes que foram modificadas e incluídas na Jikes RVM. As classes fora do pacote `ml4jit.rvm` são as que já pertenciam ao código fonte da máquina virtual.

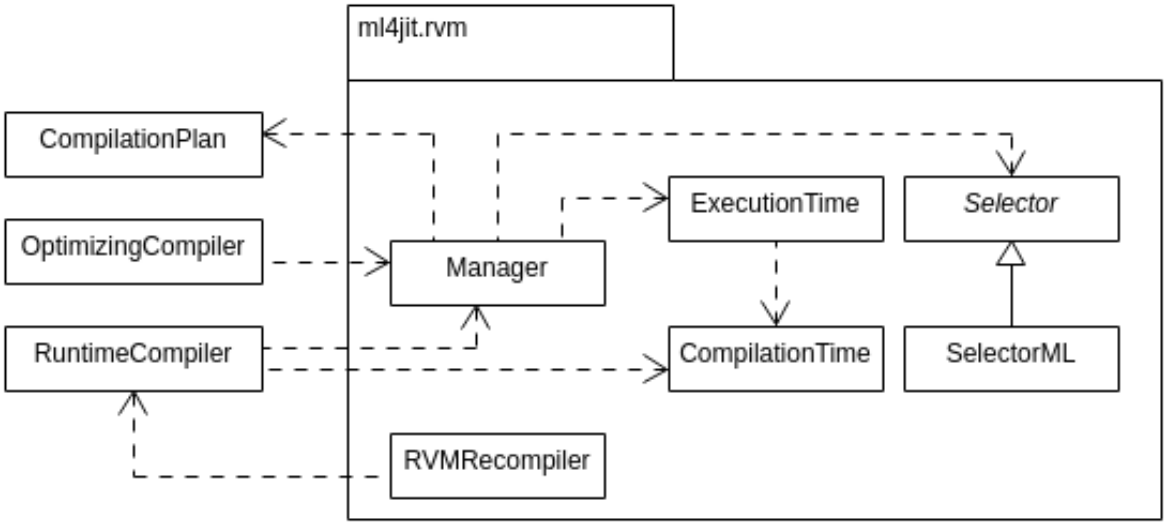

Figura 21: Diagrama da classes adicionadas e modificadas na Jikes RVM.

A classe `Manager` é responsável por gerenciar o processo de seleção de planos de compilação específico que serão aplicados a cada método do programa. Ela é inicializada pela classe `RuntimeCompiler` no processo de *boot* da máquina virtual. Ao ser inicializada, ela cria uma instância da classe `ExecutionTime`, responsável por calcular o tempo de execução do programa através da diferença do instante de tempo do início da execução do programa, quando o método `main` é acionado, pelo instante de tempo do fim do programa.

Durante a execução do programa, quando a máquina virtual decide compilar um método ela aciona a classe `RuntimeCompiler`. Caso a Jikes RVM esteja operando em modo de otimização, a classe `OptimizingCompiler` é acionada pela `RuntimeCompiler` para realizar a compilação do método. Essa classe foi modificada para que no momento da compilação do método, quando a máquina virtual estiver operando no modo *Machine Learning*, ela acione a `Manager` solicitando o plano de compilação que deve ser aplicado ao método.

Ao ser acionada, a classe `Manager` solicita a uma instância da classe abstrata `Selector` que selecione o plano de compilação para o método. Definido o plano de compilação

a classe `Manager` cria uma instância da classe `CompilationPlan` e a informa para a `OptimizingCompiler`, que realiza o processo de compilação do método de acordo plano de compilação especificado. Após o método ser compilado, a classe `RuntimeCompiler` registra o seu tempo de compilação acionando a classe `CompilationTime`. Ao final da execução do programa a classe `ExecutionTime` é acionada para criar um arquivo contendo o tempo de execução do programa; ela também aciona a classe `CompilationTime` para criar um arquivo contendo o tempo de compilação de cada método do programa.

Uma instância da classe abstrata `Selector` é utilizada para selecionar as otimizações que devem ser habilitadas em um plano de compilação para a compilação do método. Essa instância é criada na classe `Manager` através do mecanismo de *reflection* disponível na linguagem Java. A classe a ser instanciada é informada no parâmetro de configuração *SELECTOR* adicionado a Jikes RVM. Decidiu-se por este mecanismo por ele permitir que novas formas de seleção de otimizações possam ser facilmente incorporadas ao arcabouço.

A classe `SelectorML` estende a classe abstrata `Selector` e é responsável por retornar as otimizações que devem ser aplicadas de forma específica a cada método do programa. Essas otimizações são obtidas pelo recurso de *Machine Learning* do arcabouço, utilizando-se o modo estático (*offline*). Nesse modo, é gerado um arquivo contendo o conjunto de otimizações que deve ser aplicado a cada método. Esse arquivo deve ser especificado no parâmetro de configuração *ML_FILE* adicionado a Jikes RVM e carregado na inicialização da classe `SelectorML`.

A classe `RVMRecompiler` permite solicitar à máquina virtual a recompilação de um determinado método. Ela pode ser utilizada para a implementação do recurso de aprendizado dinâmico. A ideia desse recurso é que diferentes conjuntos de otimizações para um mesmo método sejam testados durante uma única execução do programa. Para isso, é necessário que o método seja recompilado a cada vez que se quer testar um novo conjunto de otimizações. Ao receber a solicitação de recompilação de um método a `RVMRecompiler` aciona a classe `RuntimeCompiler` requisitando a compilação do método. O procedimento para recompilação é o mesmo adotado para a compilação de um método, conforme descrito anteriormente.

Além da modificação do código fonte da Jikes RVM, foi necessário alterar o arquivo `ValueOptions.opt.dat` para a inclusão de novos parâmetros de configuração para a máquina virtual. Os parâmetros de configuração adicionados foram: *selector*, *ml_file* e *mode*. O parâmetro *selector* indica para a máquina virtual qual instância da classe `Selector` deve ser criada pelo mecanismo de *reflection*. O parâmetro *ml_file* indica o

caminho do arquivo que deve ser carregado quando a classe `SelectorML` é inicializada.

O parâmetro *mode* indica para a máquina virtual em qual modo ela deve operar. As opções são: *default*, *training* e *ml*. No modo *default* a máquina virtual opera de forma padrão, sem utilizar nenhuma classe definida no pacote `ml4jit.rvm`. No modo *training*, utilizado no processo de treinamento do algoritmos de aprendizado de máquina, o tempo de compilação de cada método e o tempo de execução do programa são coletados. Nesse modo, é aplicado o mesmo conjunto de otimizações para todos os métodos do programa. Esse conjunto de otimizações deve ser especificado nos parâmetros de configuração da Jikes RVM. No modo *ml*, além da coleta do tempo de compilação de cada método e o tempo de execução do programa, é aplicado um conjunto de otimizações específico para cada método do programa.

5.4 Arquitetura para Aprendizado Dinâmico

Esta seção apresenta uma proposta de arquitetura que possibilita que o arcabouço realize pesquisas experimentais envolvendo um processo de aprendizado dinâmico. O objetivo com o uso desse tipo de aprendizado, conforme descrito na seção 4.6, é permitir que o modelo de aprendizado seja alimentado com novos dados durante a execução do programa, onde diferentes conjuntos de otimizações podem ser testados para um mesmo método do programa. A Figura 22 apresenta o diagrama de classes da arquitetura proposta para aprendizado dinâmico.

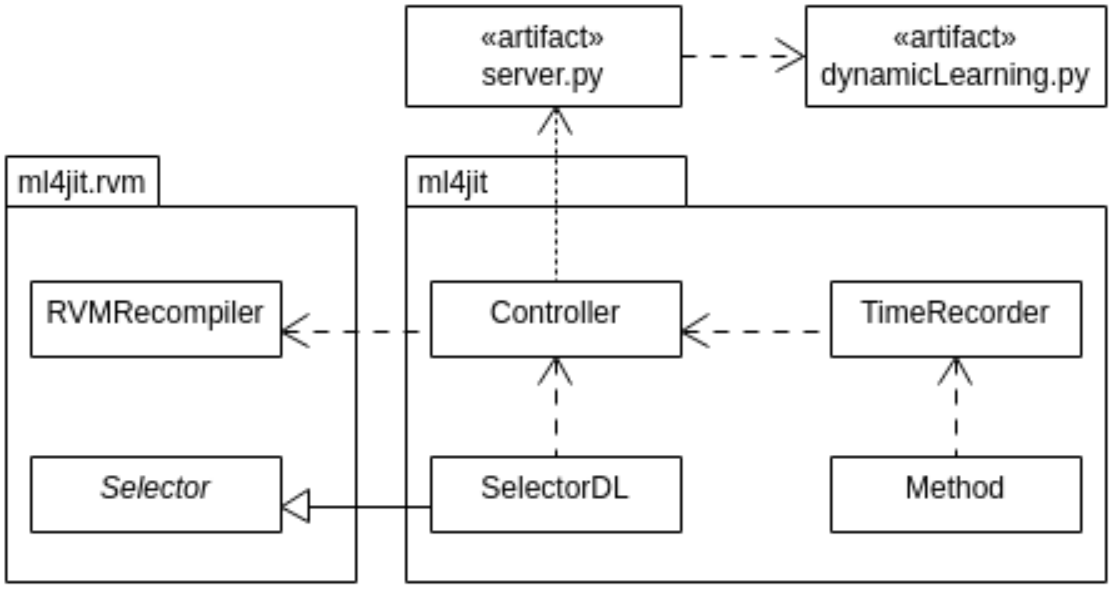

Figura 22: Diagrama de classes da arquitetura proposta para aprendizado dinâmico.

O processo de aprendizado dinâmico do arcabouço é controlado por classes presentes no agente externo e que estão contidas no arquivo `ml4jit.jar`, apresentadas na figura dentro do pacote `ml4jit`. Entretanto, para que o processo seja realizado é necessário acionar a classe `RVMRecompiler` presente na versão modificada da Jikes RVM. Portanto, existe uma dependência entre classes contidas no `ml4jit.jar` com classes contidas na versão modificada da Jikes RVM. Além disso, *scripts* são utilizados para a implementação do modelo de aprendizado dinâmico.

A classe `Controller` é responsável por decidir se um determinado método deve ser recompilado durante a execução do programa. Para auxiliar no processo de decisão, ela recebe informações da classe `TimeRecorder` que coleta os tempos de execução de cada método, conforme descrito na seção 5.1.2. Quando a `Controller` decidir recompilar um determinado método ela aciona a classe `RVMRecompiler`, presente na versão modificada da Jikes RVM, que dispara o processo de recompilação na máquina virtual, conforme descrito na seção anterior.

No momento da recompilação do método a Jikes RVM solicita a uma instância da classe abstrata `Selector` que indique qual o conjunto de otimizações deve ser aplicado ao método. Uma nova implementação da classe abstrata `Selector` foi criada para permitir que o conjunto de otimizações seja selecionado de acordo com os procedimentos para o aprendizado dinâmico. Essa classe denominada `SelectorDL` está contida no arquivo `ml4jit.jar` e deve ser informada para Jikes RVM pelo parâmetro *selector*, descrito na seção anterior, para permitir que pelo mecanismo de *reflection* seja criada uma instância dessa classe.

Quando a classe `SelectorDL` for acionada com uma mensagem para selecionar o conjunto de otimizações que deve ser aplicado ao método que será compilado, ela chama a `Controller` que, nesse momento, comunica-se com o modelo de aprendizado dinâmico, solicitando o conjunto de otimizações que deve ser utilizado. O modelo de aprendizado dinâmico é implementado através de *scripts* escritos na linguagem Python versão 2.7. A comunicação entre a `Controller` e o modelo de aprendizado dinâmico é feita através de um servidor, utilizando uma conexão *socket*.

Após o método ser compilado e os dados de seus tempos de execução e compilação serem coletados, a `Controller` envia esses dados, via servidor, ao modelo de aprendizado dinâmico. Esses novos dados são utilizados para agregar informações ao modelo de aprendizado e permitir que ele tome decisões com mais acurácia.

A realização do modelo de aprendizado dinâmico pode ser feita utilizando-se, por

exemplo, aprendizado ativo (SETTLES, 2012) ou aprendizado por reforço (SUTTON; BARTO, 2017). Estudos mais aprofundados devem ser realizados para adequar esses tipos de aprendizado ao problema de selecionar o melhor conjunto de otimizações para cada método de um programa.

5.5 Método de Uso do Arcabouço

Esta seção apresenta uma visão geral do método de utilização do arcabouço `ML4JIT` [3] para a realização dos experimentos de pesquisas na tentativa de descobrir o melhor conjunto de otimizações específico para métodos de programas Java utilizando aprendizado de máquina. Os parâmetros de configuração para a realização dos experimentos estão definidos no arquivo `config.py`. Os programas utilizados nos experimentos estão definidos no arquivo `benchmarks.py`. Ele contém alguns programas pré-definidos, entretanto, é possível que novos programas sejam adicionados ao arcabouço.

A realização dos experimentos é dividida em duas etapas: captura dos dados para treinamento do modelo de aprendizado de máquina e teste com o modelo de aprendizado na seleção do conjunto de otimizações específico para cada método de um programa Java. Ao final desta etapa, são obtidos dados utilizados para a análise do desempenho dos programas testados.

A etapa de captura dos dados para treinamento é realizada com a execução do arquivo `training.py`. Este arquivo recebe como argumento o conjunto de programas (*benchmark*) e o nível de otimização que serão analisados. Para cada programa do *benchmark*, são extraídas as características de seus métodos e, após essa extração, o programa é executado diversas vezes. A cada execução, um determinado conjunto de otimizações do nível selecionado é utilizado. O arcabouço possibilitar analisar todas as permutações do conjunto de otimizações de um determinado nível ou analisar um determinado número de permutações, selecionadas aleatoriamente. Entretanto, sempre uma execução do programa é feita com todas as opções de um determinado nível de otimização habilitadas. Os dados obtidos com essa execução são usados como referência para a comparação com as demais execuções.

A cada execução do programa é obtido o tempo de execução de cada método do programa para cada conjunto de otimizações analisado naquela execução. O tempo de execução de cada método do programa executado com um determinado conjunto de oti-

[3] O manual do usuário do arcabouço `ML4JIT` está disponível em: https://github.com/amignon/ml4jit.git.

mizações é então comparado com o tempo de execução obtido para o método quando o programa é executado com o conjunto de otimizações de referência. Se o tempo de execução do método utilizando um conjunto de otimizações for superior a um percentual, definido no arquivo de configuração, em relação ao tempo de execução de referência do método, a descrição deste conjunto de otimizações é associada ao vetor de características do método. Ao final, é gerado, para cada programa analisado, um arquivo contendo as características dos métodos e os conjuntos de otimizações associados a elas.

A etapa de teste com o modelo de aprendizado é realizada com a execução do arquivo `experiment.py`. Este arquivo recebe como argumento o conjunto de programas (*benchmark*) e o nível de otimização que serão testados. Além disso, deve-se definir, no arquivo de configuração, qual algoritmo de aprendizado de máquina deve ser utilizado para no teste. Para cada programa do *benchmark* que será testado, é gerado um modelo de aprendizado de máquina utilizando a técnica *leave-one out cross validation* (MURPHY, 2012). O modelo de aprendizado de máquina gerado é então utilizado para selecionar qual conjunto de otimização deve ser aplicado a cada método do programa, a partir das características definidas para o método.

Para obter os resultados do teste do modelo de aprendizado, cada programa do *benchmark* é executado de duas formas diferentes: uma em que todas as otimizações do nível selecionado estão habilitadas e outra com as otimizações definidas pelo modelo de aprendizado. Como resultado de cada execução, são obtidos a soma do tempo de compilação de cada método do programa e o tempo de execução total do programa. Por questões estatísticas, as duas formas de execução do programa são rodadas diversas vezes. O número de vezes é definido no arquivo de configuração. Com isso, pode-se a média ou a mediana dos tempos de compilação e execução dos programas de cada uma das formas de execução.

Ao final da etapa de teste com o modelo de aprendizado, é gerado, para cada programa analisado, um arquivo contendo a lista dos tempos de compilação e execução do programa obtidos nas duas formas de execução. Esse arquivo é utilizado pelo programa *result.py* para a geração dos gráficos comparativos dos resultados do experimento.

5.6 Conclusão

Este capítulo apresentou os detalhes de projeto e implementação do arcabouço `ML4JIT`. Apresentou-se também uma visão geral de seu método de utilização. O próximo capítulo apresenta os resultados dos experimentos realizados neste trabalho para a validação do projeto e da implementação do arcabouço.

6 EXPERIMENTOS E RESULTADOS

Este capítulo apresenta os experimentos realizados para a validação do projeto e da implementação do arcabouço `ML4JIT`. O objetivo dos experimentos é o de verificar se com o uso de aprendizado de máquina é possível diminuir o tempo de execução e/ou o tempo de compilação dos programas analisados.

6.1 Infraestrutura

Os experimentos foram executados em uma máquina com processador Intel Core i5-2410M de 2.30GHz com 4GB de memória RAM, 64KB de cache L1, 256KB de cache L2 e 3MB de cache L3, rodando Linux Mint versão 17.3 64 bits, Java versão 1.6.0_40 e Python versão 2.7.6. A Tabela 6 apresenta as bibliotecas que auxiliaram no desenvolvimento e execução dos *scripts*.

Tabela 6: Bibliotecas Python utilizadas no desenvolvimento e execução dos *scripts*.

Biblioteca	**Versão**
matplotlib	2.0.0
numpy	1.12.1
pandas	0.18.1
scipy	0.13.3
scikit-learn	0.18.1

A máquina virtual utilizada foi a Jikes Research Virtual Machine (Jikes RVM) versão 3.1.4. Utilizou-se a configuração `production`, o que indica que o núcleo da máquina virtual foi compilado pelo compilador otimizador. O sistema adaptativo da Jikes RVM foi desabilitado.

Os tempos de compilação e de execução do programa foram obtidos diretamente da Jikes RVM, através de arquivos gerados pela máquina virtual. O tempo de execução foi calculado pela diferença do instante de tempo obtido no instante em que o programa inicia sua execução na máquina virtual, isto é, quando o método `main` do programa é acionado,

pelo instante de tempo obtido ao final da execução do programa.

6.2 Benchmarks

Os experimentos foram executados utilizando um subconjunto de programas presentes nos *benchmarks*: Java Grande Forum Suite v2.0 (JGF) (BULL et al., 2000) e DaCapo 9.12 (BLACKBURN et al., 2006). A seguir, são apresentadas tabelas com a descrição dos programas utilizados de cada *benchmark*. Alguns programas dos *benchmarks* foram retirados dos experimentos por apresentarem problemas durante sua execução. Eles podem ter ocorrido devido a Jikes RVM não ter suporte para um determinado tipo de programa ou o código inserido no processo de instrumentalização dos métodos causar algum tipo de efeito colateral. É necessária uma análise mais aprofundada para descobrir a causa dos problemas apresentados.

A Tabela 7 apresenta a descrição dos programas utilizados do *benchmark* JGF - Section 2. Os experimentos foram executados utilizando a versão *Size B* desses programas. A Tabela 8 apresenta a descrição dos programas utilizados do *benchmark* DaCapo 9.12. Os experimentos foram executados com o parâmetro `-s small`.

Tabela 7: Descrição dos programas do *benchmark* JGF - Section 2. Adaptada de (BULL et al., 2000).

Programa	**Descrição**
Series	Computes the first N Fourier coecients of the function $f(x) = (x + 1)^x$ on the interval 0,2.
LUFact	Solves an $N \times N$ linear system using LU factorisation followed by a triangular solve.
SOR	Performs 100 iterations of successive over-relaxation on an $N \times N$ grid.
HeapSort	Sorts an array of N integers using a heap sort algorithm.
Crypt	Performs IDEA encryption and decryption on an array of N bytes
FFT	Performs a one-dimensional forward transform of N complex numbers.
Sparse	Uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure

6.3 Método Empregado nos Experimentos

Esta seção apresenta o método empregado para a execução dos experimentos. O objetivo é comparar os tempos de execução e de compilação dos programas utilizando um compilador que compila cada método do programa com um conjunto de otimizações

Tabela 8: Descrição dos programas do *benchmark* DaCapo.

Programa	**Descrição**
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.
pmd	Analyzes a set of Java classes for a range of source code problems.
sunflow	Renders a set of images using ray tracing.

específico, baseado nas heurísticas obtidas dos algoritmos de aprendizado de máquina, com um compilador sem modificação.

Dois tipos de experimentos foram realizados: um com o nível de otimização O0 e outro com o nível de otimização O1 da Jikes RVM. Todos os métodos dos programas utilizados foram compilados usando somente as otimizações presentes em cada nível de otimização. As heurísticas obtidas dos algoritmos de aprendizado de máquina indicam um subconjunto dessas otimizações que serão aplicadas em cada método que é compilado. Nos dois tipos de experimentos, foram utilizados dois algoritmos diferentes de aprendizado de máquina supervisionado, presentes na biblioteca *scikit-learn* (PEDREGOSA et al., 2011): o classificador *DecisionTreeClassifier* e o classificador *KNeighborsClassifier*. Os valores dos parâmetros de configuração adotados são os sugeridos por padrão pela biblioteca.

A tabela 9 apresenta as otimizações utilizadas em cada nível, sendo que as do nível O1 incluem todas do nível O0. As otimizações de *inline* e de reordenação de código foram desabilitadas nos experimentos, em ambos os níveis de otimização. Devido a forma de instrumentalização do *bytecode* dos métodos, a aplicação dessas transformações pode modificar o comportamento do código para a medição de tempo do método. Um estudo mais aprofundado dessas otimizações deve ser realizado para a identificação dos possíveis problemas que possam apresentar.

Cada um dos *benchmarks* apresentados na seção anterior foram analisados utilizando a técnica *leave-one out cross validation* (MURPHY, 2012). Dado o conjunto de n programas de um *benchmark*, são utilizados $n - 1$ programas para o treinamento do modelo de aprendizado de máquina. No outro programa, denominado de programa de teste, são aplicadas as heurísticas geradas.

Os dados para o treinamento dos algoritmos de aprendizado de máquina nos experi-

Tabela 9: Otimizações da Jikes RVM utilizadas nos experimentos realizados.

Otimização	**Nível**
field_analysis	0
local_constant_prop	0
local_copy_prop	0
local_cse	0
regalloc_coalesce_moves	0
regalloc_coalesce_spills	0
control_static_splitting	1
escape_scalar_replace_aggregates	1
escape_monitor_removal	1

mentos com o nível de otimização O0 foram obtidos pela execução de cada programa do *benchmark* com todas as permutações de otimizações possíveis. Nos experimentos com o nível de otimização O1 foram utilizadas 100 permutações, incluindo a de referência - quando todas as otimizações estão habilitadas. Ao final, foram selecionados, para cada método do programa, os conjuntos de otimizações que obtiveram uma melhora de tempo de execução do método em pelo menos 1% em relação a execução de referência.

Após o treinamento do modelo de aprendizado, são geradas heurísticas para cada método do programa que será testado. Elas indicam quais otimizações devem ser aplicadas para cada método do programa. As heurísticas são então integradas ao compilador e aplicadas ao programa de teste. A próxima seção apresenta os resultados obtidos com a execução dos experimentos.

6.4 Resultados

Essa seção apresenta os resultados dos experimentos realizados para a validação do arcabouço. São comparados os dados de tempo de execução e de tempo de compilação de cada um dos programas dos *benchmarks*. A comparação é feita entre os tempos obtidos com a execução do programa com o uso de aprendizado de máquina e os tempos obtidos com a execução do programa usando a configuração padrão de um determinado nível. Os tempos foram obtidos pela mediana de 20 execuções de cada programa. Nas tabelas e figuras apresentadas, o valor 1.0 indica que a execução com uso de aprendizado de máquina obteve um desempenho igual ao da execução de referência - com a utilização da configuração padrão; e abaixo de 1.0 o desempenho foi melhor que o obtido com a execução de referência.

As informações obtidas dos algoritmos de aprendizado de máquina indicam quais

otimizações devem ser habilitadas ou desabilitadas para cada um dos métodos do programa. A Tabela 10 apresenta um exemplo das informações retornadas pelos algoritmos de aprendizado de máquina utilizados nos experimentos para o método *SparseMatmult.test* do programa *Sparse* do *benchmark* JGF - Section 2 no nível de otimização O0.

Tabela 10: Informações retornadas pelos algoritmos de aprendizado de máquina utilizados nos experimentos indicando quais otimizações devem ser habilitadas (1) ou desabilitadas (0) para o método *SparseMatmult.test* do programa *Sparse* do *benchmark* JGF - Section 2 no nível de otimização O0.

Otimização	**Padrão**	**DecisionTreeClassifier**	**KNeighborsClassifier**
field_analysis	1	1	0
local_constant_prop	1	0	1
local_copy_prop	1	1	1
local_cse	1	1	1
regalloc_coalesce_moves	1	1	0
regalloc_coalesce_spills	1	1	0

6.4.1 Nível de Otimizaçao O0

A Tabela 11 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* JGF - Section 2 utilizando o classificador *DecisionTreeClassifier*. A Tabela 12 e a Figura 23 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), houve um ganho no desempenho no tempo de compilação em aproximadamente 4.3% porém, o tempo de execução obteve um desempenho inferior em aproximadamente 2.3%, sendo que o programa *HeapSort* pode ter impactado nesse resultado, já que seu desempenho no tempo de execução foi aproximadamente 17.8% inferior ao tempo de execução da configuração padrão. O programa que teve a maior redução em relação ao tempo de execução foi o *Sparse*, com redução de aproximadamente 1.1%. O programa *LUFact* apesar de apresentar desempenho superior no tempo de execução em aproximadamente 0.4% apresentou um desempenho inferior no tempo de compilação em aproximadamente 0.8%.

Tabela 11: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* JGF - Section 2 no nível de otimização O0 utilizando o classificador *DecisionTreeClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
Sparse	1119.50	100.64	1131.50	89.64	29.73	7.51	32.57	6.61
Crypt	3710.00	31.29	3711.50	33.35	40.69	9.03	42.83	11.08
LUFact	1403.50	24.47	1408.50	26.06	43.15	7.80	42.77	8.99
HeapSort	2361.00	26.31	2004.00	36.22	31.06	6.18	32.43	8.91
FFT	36428.50	364.83	35930.50	372.99	32.13	4.96	32.83	5.27
SOR	1795.00	11.98	1796.00	17.53	30.87	7.08	33.11	5.87
Series	34801.50	87.17	34803.00	70.10	32.91	8.95	34.03	11.34

Tabela 12: Dados do desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O0 utilizando o classificador *DecisionTreeClassifier*.

Program	**Execution**	**Compilation**
Sparse	0.989395	0.912794
Crypt	0.999596	0.950058
LUFact	0.996450	1.008954
HeapSort	1.178144	0.957779
FFT	1.013860	0.978762
SOR	0.999443	0.932306
Series	0.999957	0.967164
geo	1.023483	0.957822

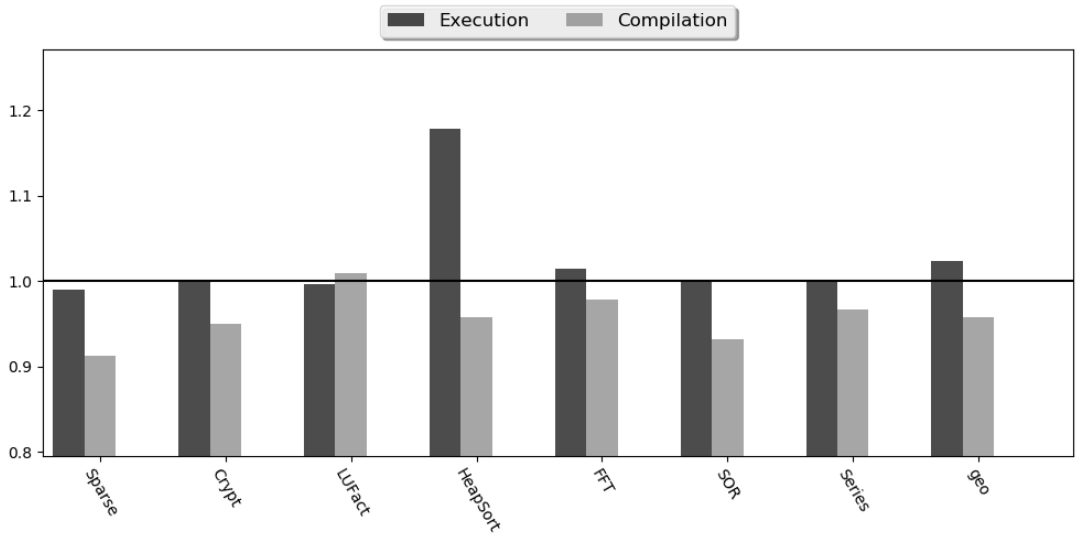

Figura 23: Desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O0 utilizando o classificador *DecisionTreeClassifier*.

A Tabela 13 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* JGF - Section 2 utilizando o classificador *KNeighborsClassifier*. A Tabela 14 e a Figura 24 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), o desempenho no tempo de execução foi superior em aproximadamente 0.8% e o desempenho no tempo de compilação melhorou em aproximadamente 1.2%, em relação a configuração padrão. O programa *HeapSort* merece destaque, pois o seu tempo de compilação obteve um desempenho inferior de aproximadamente 1% enquanto o seu tempo de execução obteve um desempenho superior de aproximadamente 4.3%.

Tabela 13: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* JGF - Section 2 no nível de otimização O0 utilizando o classificador *KNeighborsClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
Sparse	1172.00	122.67	1176.50	142.48	30.59	6.70	31.13	7.35
Crypt	3692.50	29.92	3714.00	47.36	40.91	10.71	42.09	11.29
LUFact	1405.50	17.29	1407.00	33.88	40.83	6.22	42.43	9.09
HeapSort	1909.50	28.58	1993.50	16.45	31.73	7.90	31.41	2.69
FFT	36248.50	311.59	36159.00	337.14	32.70	5.93	32.94	6.30
SOR	1790.50	15.16	1790.50	17.06	28.72	5.73	29.32	6.76
Series	34770.00	86.43	34808.50	88.88	34.14	7.23	33.49	8.50

Tabela 14: Dados do desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O0 utilizando o classificador *KNeighborsClassifier*.

Program	**Execution**	**Compilation**
Sparse	0.996175	0.982631
Crypt	0.994211	0.971918
LUFact	0.998934	0.962186
HeapSort	0.957863	1.010369
FFT	1.002475	0.992688
SOR	1.000000	0.979636
Series	0.998894	1.019371
geo	0.992544	0.988218

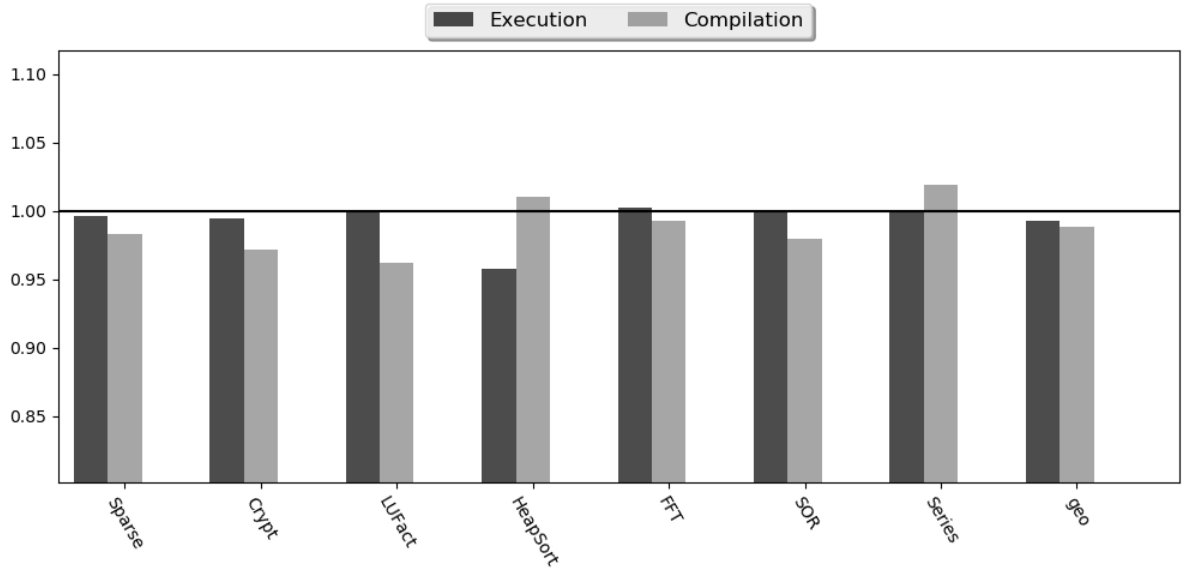

Figura 24: Desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O0 utilizando o classificador *KNeighborsClassifier*.

A Tabela 15 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* DaCapo utilizando o classificador *DecisionTreeClassifier*. A Tabela 16 e a Figura 25 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), o desempenho com o uso de aprendizado de máquina foi inferior em relação a configuração padrão. O desempenho no tempo de execução foi pior em aproximadamente 1.6% e o desempenho no tempo de compilação foi inferior em aproximadamente 1%. O desempenho de nenhum dos programas no tempo de execução foi superior ao da configuração padrão. Entretanto, os programas *h2* e *luindex* obtiveram um desempenho superior no tempo de compilação em aproximadamente 1.8% e 0.3%, respectivamente.

Tabela 15: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* DaCapo no nível de otimização O0 utilizando o classificador *DecisionTreeClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
lusearch	2622.50	80.79	2613.50	108.46	1790.37	61.11	1779.72	87.65
fop	18380.00	240.09	17386.50	169.66	17327.58	236.68	16370.95	171.73
avrora	5263.50	87.31	5185.00	85.46	2048.73	59.20	2044.49	47.75
sunflow	3088.00	73.61	3020.50	92.71	2044.95	57.74	2000.30	69.34
h2	27836.50	4036.62	27767.50	4630.44	4440.34	457.42	4517.87	536.02
luindex	2842.50	88.00	2815.50	53.84	1998.45	34.50	2003.32	37.94
pmd	3314.00	66.10	3283.00	57.76	2753.72	54.95	2730.50	47.20

Tabela 16: Dados do desempenho dos programas do *benchmark* DaCapo no nível de otimização O0 utilizando o classificador *DecisionTreeClassifier*.

Program	**Execution**	**Compilation**
lusearch	1.003444	1.005982
fop	1.057142	1.058434
avrora	1.015140	1.002074
sunflow	1.022347	1.022323
h2	1.002485	0.982839
luindex	1.009590	0.997570
pmd	1.009443	1.008504
geo	1.016936	1.010863

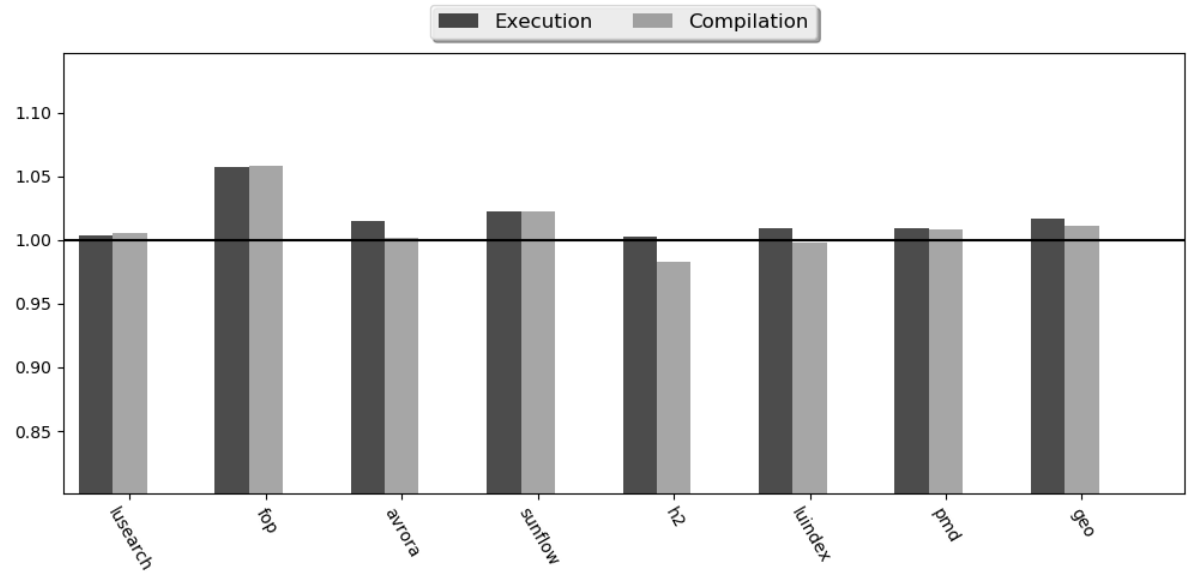

Figura 25: Desempenho dos programas do *benchmark* DaCapo no nível de otimização O0 utilizando o classificador *DecisionTreeClassifier*.

A Tabela 17 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* DaCapo utilizando o classificador *KNeighborsClassifier*. A Tabela 18 e a Figura 26 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), com o uso de aprendizado de máquina houve um ganho no desempenho tanto no tempo de execução quanto no tempo de compilação, em relação a configuração padrão. O desempenho no tempo de execução foi superior em aproximadamente 3.5% e o desempenho no tempo de compilação foi melhor em aproximadamente 3.4%. O programa *fop* merece destaque por ter apresentado um desempenho melhor no tempo de execução em aproximadamente 22% enquanto o desempenho do seu tempo de compilação foi superior em aproximadamente 23.7%.

Tabela 17: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* DaCapo no nível de otimização O0 utilizando o classificador *KNeighborsClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
lusearch	2646.50	85.63	2620.00	86.89	1807.05	73.31	1778.78	71.25
fop	13460.50	73.36	17152.00	221.12	12407.53	70.44	16142.88	217.39
avrora	5305.50	88.13	5188.50	95.85	2078.21	51.82	2022.44	36.99
sunflow	3106.50	54.56	3047.50	91.56	2060.45	42.52	2025.30	71.53
h2	25602.00	4470.83	26501.00	4995.65	4459.32	488.26	4470.31	613.78
luindex	2848.50	121.45	2825.00	44.88	2010.32	38.23	2002.31	29.97
pmd	3239.50	41.11	3325.50	62.01	2657.17	36.48	2756.05	49.95

Tabela 18: Dados do desempenho dos programas do *benchmark* DaCapo no nível de otimização O0 utilizando o classificador *KNeighborsClassifier*.

Program	**Execution**	**Compilation**
lusearch	1.010115	1.015894
fop	0.784777	0.768607
avrora	1.022550	1.027575
sunflow	1.019360	1.017353
h2	0.966077	0.997542
luindex	1.008319	1.004002
pmd	0.974139	0.964125
geo	0.965847	0.966571

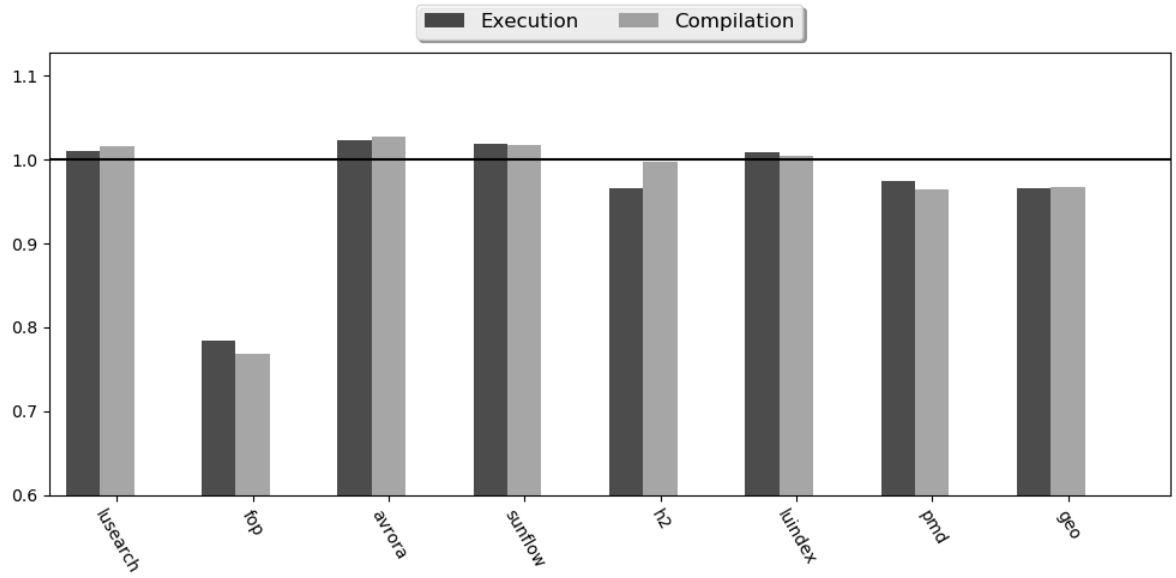

Figura 26: Desempenho dos programas do *benchmark* DaCapo no nível de otimização O0 utilizando o classificador *KNeighborsClassifier*.

6.4.2 Nível de Otimizaçao O1

A Tabela 19 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* JGF - Section 2 utilizando o classificador *DecisionTreeClassifier*. A Tabela 20 e a Figura 27 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), o desempenho no tempo de compilação foi melhor em aproximadamente 3.5% porém, o desempenho no tempo de execução foi inferior em aproximadamente 1.2%, sendo que o programa *LUFact* pode ter impactado nesse resultado, já que seu desempenho no tempo de execução foi aproximadamente 19% inferior ao tempo de execução da configuração padrão. O programa que obteve um melhor desempenho no tempo de execução foi o *HeapSort* em aproximadamente 5.5%.

Tabela 19: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* JGF - Section 2 no nível de otimização O1 utilizando o classificador *DecisionTreeClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
Sparse	1144.00	104.10	1171.50	107.62	32.77	5.81	33.80	6.22
Crypt	3722.00	24.96	3721.50	24.22	44.71	8.38	46.06	10.20
LUFact	1762.50	24.53	1474.00	32.91	45.25	10.14	46.06	11.20
HeapSort	1892.00	28.38	1998.00	22.74	32.62	4.69	34.19	5.58
FFT	36034.00	363.47	36503.00	401.78	34.74	6.46	35.50	7.40
SOR	1788.50	14.69	1789.50	20.79	32.13	7.71	34.70	8.89
Series	34803.50	221.57	34805.00	75.24	35.68	11.19	36.30	9.06

Tabela 20: Dados do desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O1 utilizando o classificador *DecisionTreeClassifier*.

Program	**Execution**	**Compilation**
Sparse	0.976526	0.969803
Crypt	1.000134	0.970716
LUFact	1.195726	0.982390
HeapSort	0.946947	0.954207
FFT	0.987152	0.978524
SOR	0.999441	0.925979
Series	0.999957	0.982912
geo	1.012519	0.966175

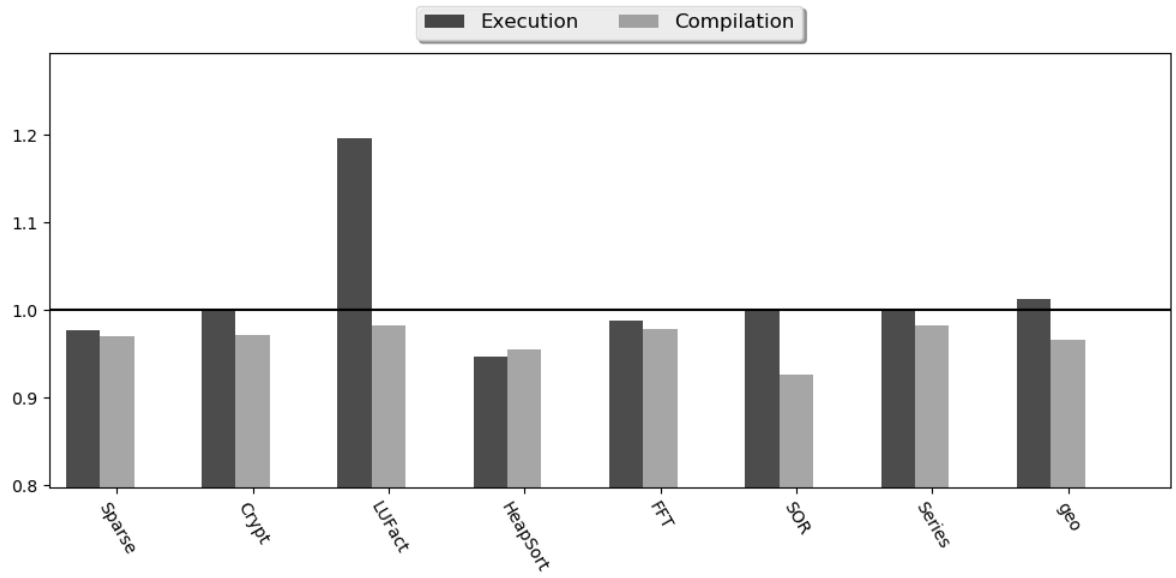

Figura 27: Desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O1 utilizando o classificador *DecisionTreeClassifier*.

A Tabela 21 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* JGF - Section 2 utilizando o classificador *KNeighborsClassifier*. A Tabela 22 e a Figura 28 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), o resultado do tempo de execução obtido com o uso de aprendizado de máquina foi similares ao obtido com o conjunto de otimizações padrão. O desempenho do tempo de execução foi superior em aproximadamente 0.2% e o desempenho no tempo de compilação foi superior em aproximadamente 3.6%. O programa que obteve um melhor desempenho no tempo de execução foi o *Sparse* em aproximadamente 3.6% sendo que seu desempenho no tempo de compilação foi superior em aproximadamente 8.4%.

Tabela 21: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* JGF - Section 2 no nível de otimização O1 utilizando o classificador *KNeighborsClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
Sparse	1127.50	74.02	1157.00	120.00	33.44	6.52	36.48	8.49
Crypt	3699.00	27.60	3730.50	38.01	45.56	7.37	46.81	10.70
LUFact	1486.00	41.65	1475.50	26.08	46.90	10.71	46.67	4.61
HeapSort	2006.50	34.42	2003.00	24.60	33.32	6.59	34.09	6.76
FFT	36380.00	367.80	35961.00	365.06	34.83	5.65	35.19	4.79
SOR	1784.00	4.67	1784.00	8.78	30.56	2.02	31.85	5.62
Series	34801.50	87.62	34818.50	211.13	35.58	8.68	35.58	8.40

Tabela 22: Dados do desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O1 utilizando o classificador *KNeighborsClassifier*.

Program	**Execution**	**Compilation**
Sparse	0.974503	0.916616
Crypt	0.991556	0.973291
LUFact	1.007116	1.005066
HeapSort	1.001747	0.977420
FFT	1.011652	0.989872
SOR	1.000000	0.959488
Series	0.999512	1.000266
geo	0.997949	0.974169

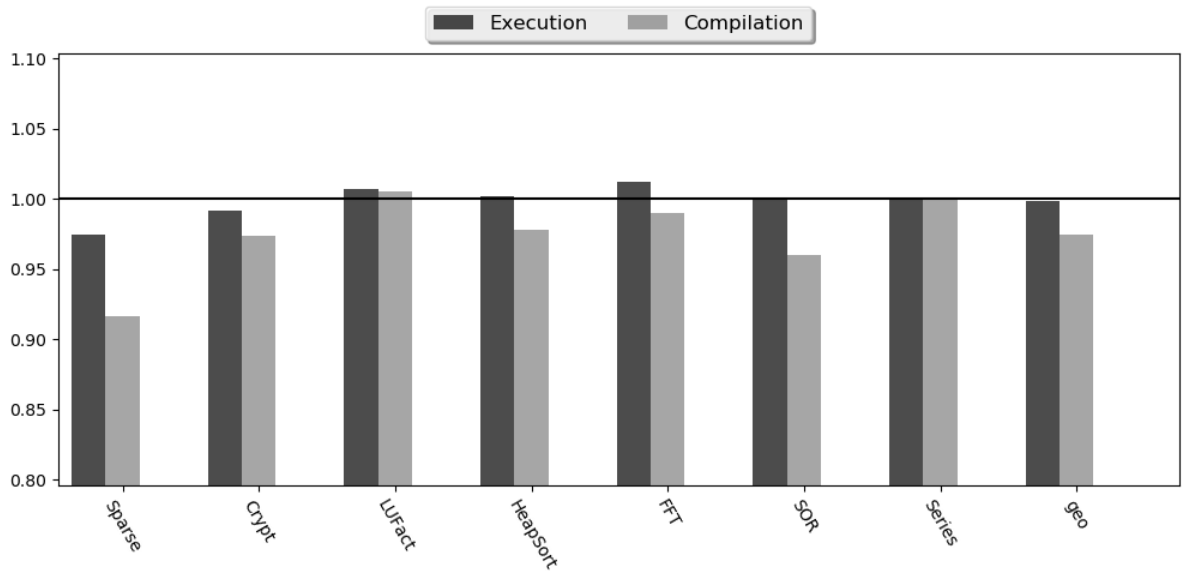

Figura 28: Desempenho dos programas do *benchmark* JGF - Section 2 no nível de otimização O1 utilizando o classificador *KNeighborsClassifier*.

A Tabela 23 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* DaCapo utilizando o classificador *DecisionTreeClassifier*. A Tabela 24 e a Figura 29 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), o desempenho com o uso de aprendizado de máquina foi superior ao obtido com o conjunto de otimizações padrão. O desempenho no tempo de execução foi superior em aproximadamente em 3.7% e o tempo de compilação foi melhor em aproximadamente 4.8%. O programa *fop* merece destaque por ter apresentado um desempenho superior no tempo de execução em aproximadamente 23% enquanto seu desempenho no tempo de compilação foi melhor em aproximadamente 26%. Entretanto, o programa *h2* apresentou desempenho inferior tanto no tempo de execução quanto no tempo de compilação em aproximadamente 2.7% e 1.7%, respectivamente.

Tabela 23: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* DaCapo no nível de otimização O1 utilizando o classificador *DecisionTreeClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
lusearch	2686.50	84.11	2721.00	84.60	1862.87	73.20	1886.02	69.16
fop	7631.00	34.38	9899.50	80.55	6566.47	33.27	8862.97	75.35
avrora	5364.50	69.14	5309.50	77.61	2201.74	47.44	2197.85	36.42
sunflow	3152.00	89.66	3190.00	75.66	2112.36	73.65	2145.84	63.63
h2	28915.50	5818.56	28129.50	4104.71	4893.75	691.48	4809.58	491.18
luindex	2938.50	57.82	2921.00	64.74	2098.06	39.26	2114.66	34.97
pmd	3450.00	25.59	3507.00	50.47	2874.19	21.69	2936.38	45.71

Tabela 24: Dados do desempenho dos programas do *benchmark* DaCapo no nível de otimização O1 utilizando o classificador *DecisionTreeClassifier*.

Program	**Execution**	**Compilation**
lusearch	0.987321	0.987725
fop	0.770847	0.740889
avrora	1.010359	1.001769
sunflow	0.988088	0.984394
h2	1.027942	1.017500
luindex	1.005991	0.992146
pmd	0.983747	0.978822
geo	0.963875	0.952840

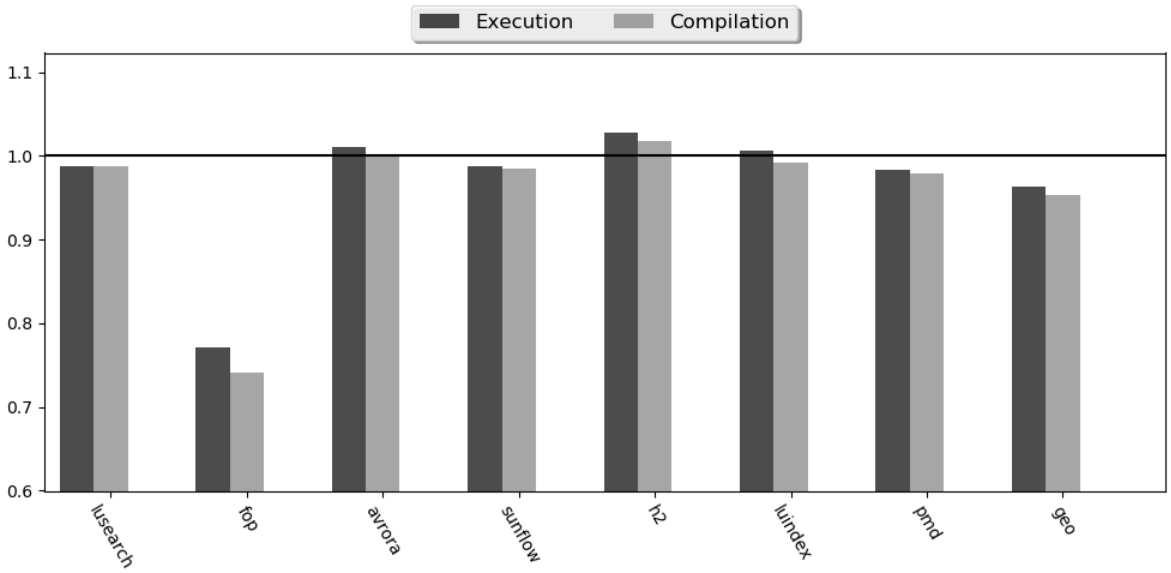

Figura 29: Desempenho dos programas do *benchmark* DaCapo no nível de otimização O1 utilizando o classificador *DecisionTreeClassifier*.

A Tabela 25 apresenta os dados da mediana dos tempos de execução e de compilação, em ms, utilizados na comparação do desempenho dos programas do *benchmark* DaCapo utilizando o classificador *KNeighborsClassifier*. A A Tabela 26 e a Figura 30 apresentam os resultados da comparação dos programas . Pode-se verificar que de forma geral, pela média geométrica (geo), o desempenho com o uso de aprendizado de máquina foi superior ao obtido com o conjunto de otimizações padrão. O desempenho no tempo de execução foi superior em aproximadamente em 2.3% e o tempo de compilação foi melhor em aproximadamente 2.8%. O programa *fop* merece destaque por ter apresentado um desempenho superior no tempo de execução em aproximadamente 11% enquanto seu desempenho no tempo de compilação foi melhor em aproximadamente 13%. O programa *avrora* apresentou desempenho similar ao da configuração parão tanto no tempo de execução quanto no tempo de compilação.

Tabela 25: Dados da mediana dos tempos de execução e de compilação, em ms, dos programas do *benchmark* DaCapo no nível de otimização O1 utilizando o classificador *KNeighborsClassifier*. A coluna *ML* apresenta a mediana dos tempos quando o programa foi executando com as heurísticas do algoritmo de aprendizado de máquina. A coluna *STD ML* apresenta o desvio padrão dessa mediana. A coluna *REF* apresenta a mediana dos tempos quando o programa foi executando com a configuração padrão. A coluna *STD REF* apresenta o desvio padrão dessa mediana.

Program	**Execution**				**Compilation**			
	ML	**STD ML**	**REF**	**STD REF**	**ML**	**STD ML**	**REF**	**STD REF**
lusearch	2715.50	67.83	2722.50	101.67	1875.41	58.09	1891.86	80.40
fop	8799.00	55.63	9856.50	70.72	7719.91	50.97	8822.83	65.57
avrora	5354.50	134.04	5317.00	86.75	2217.72	67.69	2216.88	51.85
sunflow	3141.50	61.08	3182.50	96.49	2087.86	48.33	2128.42	79.22
h2	25341.00	5819.16	26350.50	5652.79	4710.02	781.99	4847.65	739.57
luindex	2920.00	133.35	2902.50	152.26	2100.35	38.90	2103.92	49.55
pmd	3463.00	25.85	3478.00	41.14	2892.52	18.70	2905.64	34.86

Tabela 26: Dados do desempenho dos programas do *benchmark* DaCapo no nível de otimização O1 utilizando o classificador *KNeighborsClassifier*.

Program	**Execution**	**Compilation**
lusearch	0.997429	0.991307
fop	0.892710	0.874993
avrora	1.007053	1.000382
sunflow	0.987117	0.980944
h2	0.961690	0.971609
luindex	1.006029	0.998301
pmd	0.995687	0.995487
geo	0.977489	0.972367

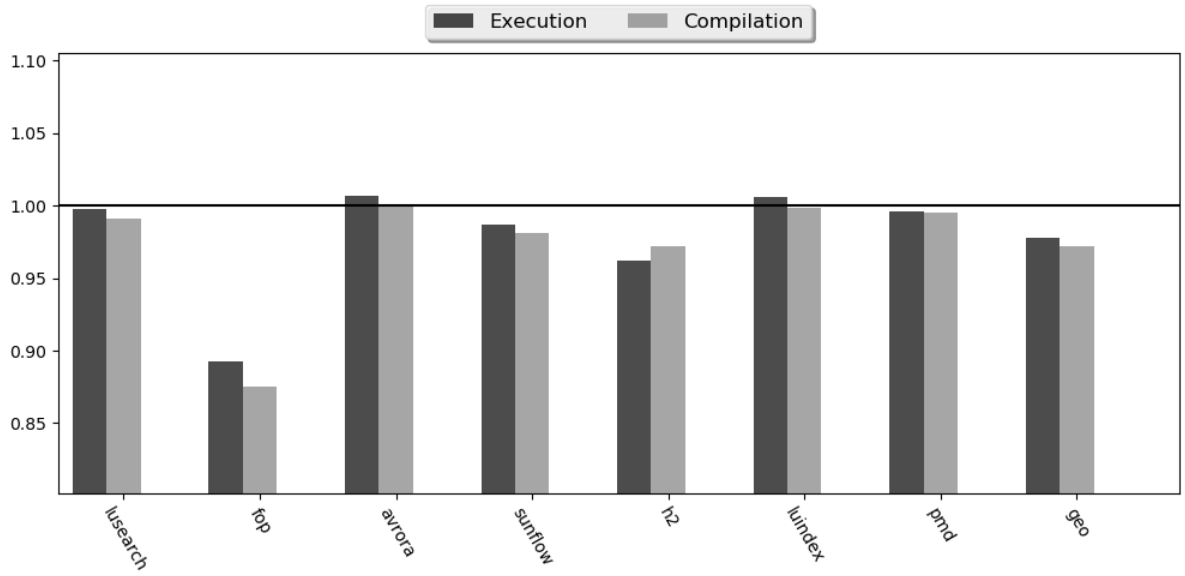

Figura 30: Desempenho dos programas do *benchmark* DaCapo no nível de otimização O1 utilizando o classificador *KNeighborsClassifier*.

6.5 Conclusão

Este capítulo apresentou experimentos iniciais realizados para a validação do projeto e da implementação do arcabouço `ML4JIT` com o objetivo de verificar se com o uso de aprendizado de máquina é possível diminuir o tempo de execução e/ou o tempo de compilação dos programas analisados. Os resultados obtidos com o uso de aprendizado de máquina foram similares aos obtidos com a configuração padrão de cada nível de otimização da Jikes RVM. Entretanto, alguns programas se beneficiaram do uso de aprendizado de máquina, tendo o seu tempo de compilação e/ou de execução reduzidos. O próximo capítulo apresenta as considerações finais deste trabalho.

7 CONSIDERAÇÕES FINAIS

Este trabalho apresentou o `ML4JIT`[1], um arcabouço de código aberto e livre para pesquisas com aprendizado de máquina em compiladores JIT para a linguagem Java. Ele permite que pesquisas experimentais, utilizando aprendizado de máquina, sejam realizadas para a descoberta do melhor conjunto de otimizações a serem aplicadas a cada método de um programa.

O arcabouço possibilita que diferentes tipos de algoritmos de aprendizado de máquina sejam avaliados, utilizando também distintos parâmetros de configuração para cada algoritmo. Com isso, pesquisas podem ser realizadas para descobrir qual o melhor tipo de algoritmo é mais adequado para o problema em questão.

Apresentou-se também uma proposta de estrutura para permitir a realização de pesquisas experimentais com aprendizado de máquina dinâmico na seleção do melhor conjunto de otimizações. Essa estrutura visa possibilitar que técnicas de aprendizado ativo ou aprendizado por reforço sejam analisadas e testadas com o arcabouço. Para diminuir o impacto do uso de aprendizado dinâmico, o arcabouço possui um recurso, baseado em *nano-patterns*, que permite identificar automaticamente métodos que possam ser excluídos do processo de instrumentalização de código.

Por ser de código aberto e livre, o `ML4JIT` pode também ser utilizado para fins didáticos em disciplinas de compiladores e de aprendizado de máquina. Em disciplinas de compiladores, o aluno pode realizar pesquisas experimentais e exercícios para analisar o impacto do uso de diferentes conjuntos de otimizações em um determinado programa. Em disciplinas de aprendizado de máquina, o aluno pode analisar o resultado do uso de diferentes tipos de algoritmos para um mesmo problema.

Embora o objetivo inicial do arcabouço seja o de tentar descobrir o melhor conjunto de otimizações que permite a diminuição do tempo de execução e/ou de compilação de um programa, ele pode ser adaptado para outros propósitos como, por exemplo, a diminuição

[1]Disponível em: https://github.com/amignon/ml4jit.git.

do consumo de energia.

7.1 Conclusão

Experimentos realizados com o uso do arcabouço `ML4JIT` validaram a sua proposta de ser um ambiente para pesquisa com aprendizado de máquina na área de compiladores JIT. Os resultados obtidos com o uso de aprendizado de máquina foram similares aos obtidos com a configuração padrão da Jikes RVM.

Entretanto, alguns programas se beneficiaram do uso de aprendizado de máquina. O que indica que com uma investigação mais aprofundada, com diferentes valores de configuração para os algoritmos e também diferentes tipos de algoritmos, pode-se tentar obter melhores resultados.

Pesquisas na área de aprendizado de máquina também podem se beneficiar do arcabouço, pois ele permite de uma forma simples, a comparação de resultados de diferentes tipos de algoritmos.

7.2 Trabalhos Futuros

Como sugestões para trabalhos futuros têm-se:

- Analisar o impacto do uso de diferentes arcabouços para a instrumentalização do *bytecode* dos métodos dos programas Java. O `ML4JIT` utiliza o arcabouço Javassist para a instrumentação de *bytecode*. Outros arcabouços como, por exemplo, ASM e BCEL podem ser analisados para verificar se há um ganho no desempenho do processo de instrumentação. Outra possibilidade é analisar se há ganhos com a realização da instrumentalização de código diretamente na máquina virtual, sem a utilização de um agente externo;

- Analisar o impacto do uso de diferentes técnicas para a medição do tempo de execução de um método (KUPERBERG; KROGMANN; REUSSNER, 2009). Por meio dessa análise pode-se encontrar uma técnica que tenha uma acurácia melhor para a medição de tempo;

- Analisar o impacto no arcabouço com a utilização de otimizações de *inline* e reordenação de código, omitidas dos experimentos realizados;

- Realizar pesquisas experimentais com aprendizado dinâmico, desenvolvendo técnicas para a aplicação de algoritmos de aprendizado ativo ou aprendizado por reforço.

REFERÊNCIAS

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2. ed. Boston: Addison-Wesley, 2006.

ALPAYDIN, E. *Introduction to Machine Learning*. 2. ed. Cambridge, MA, USA: MIT press, 2010.

ARNOLD, M. et al. Adaptive optimization in the jalapeño jvm. *ACM SIGPLAN Notices*, ACM, New York, NY, USA, v. 35, n. 10, p. 47–65, out. 2000. ISSN 0362-1340.

_____. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, IEEE, v. 93, n. 2, p. 449–466, 2005.

ARNOLD, M.; HIND, M.; RYDER, B. G. An empirical study of selective optimization. In: _____. *Languages and Compilers for Parallel Computing: 13th International Workshop, LCPC 2000 Yorktown Heights, NY, USA, August 10–12, 2000 Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 49–67. ISBN 978-3-540-45574-5.

ARNOLD, M.; RYDER, B. G. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 36, n. 5, p. 168–179, maio 2001. ISSN 0362-1340.

BALL, T.; LARUS, J. R. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 16, n. 4, p. 1319–1360, jul. 1994. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/183432.183527⟩.

BERUBE, P.; AMARAL, J. N. Combined profiling: A methodology to capture varied program behavior across multiple inputs. In: *2012 IEEE International Symposium on Performance Analysis of Systems Software*. [S.l.: s.n.], 2012. p. 210–220.

BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 0387310738.

BLACKBURN, S. M. et al. The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 2006. p. 169–190.

BREIMAN, L. Random forests. *Machine Learning*, v. 45, n. 1, p. 5–32, 2001. ISSN 1573-0565.

BULL, J. M. et al. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, John Wiley & Sons, Ltd., v. 12, n. 6, p. 375–388, 2000. ISSN 1096-9128.

CAVAZOS, J.; MOSS, J. E. B. Inducing heuristics to decide whether to schedule. *ACM SIGPLAN Notices*, ACM, v. 39, n. 6, p. 183–194, 2004.

CAVAZOS, J.; O'BOYLE, M. F. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*, ACM, v. 41, n. 10, p. 229–240, 2006.

CHIBA, S. Load-time structural reflection in java. In: *Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK, UK: Springer-Verlag, 2000. (ECOOP '00), p. 313–336. ISBN 3-540-67660-0. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=646157.679856⟩.

COOPER, K. D.; TORCZON, L. *Engineering a Compiler*. 2. ed. Burlington, MA, USA: Elsevier, 2011.

DAHM, M. *Byte Code Engineering with the BCEL API*. [S.l.], 2001.

ERTEL, W. *Introduction to Artificial Intelligence*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2011. ISBN 9780857292988.

FURSIN, G. et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, Springer, v. 39, n. 3, p. 296–327, 2011.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GIL, J. Y.; MAMAN, I. Micro patterns in java code. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 10, p. 97–116, out. 2005. ISSN 0362-1340.

GRUNE, D. et al. *Modern Compiler Design*. 2. ed. New York, NY, USA: Springer Science & Business Media, 2012.

HALL, M.; PADUA, D.; PINGALI, K. Compiler research: the next 50 years. *Communications of the ACM*, ACM, v. 52, n. 2, p. 60–67, 2009.

HOSTE, K.; GEORGES, A.; EECKHOUT, L. Automated just-in-time compiler tuning. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: ACM, 2010. (CGO '10), p. 62–72. ISBN 978-1-60558-635-9.

JOSEPH, P. et al. The compiler design handbook: Optimizations and machine code generation. In: _____. 2. ed. Boca Raton, FL, USA: CRC Press, 2007. cap. Statistical and Machine Learning Techniques in Compiler Design, p. 8:1–8:32.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, v. 4, p. 237–285, 1996.

KULESHOV, E. Using the asm framework to implement common java bytecode transformation patterns. In: *Proceedings of the 6th AOSD*. [S.l.]: ACM Press, 2007.

KULKARNI, S.; CAVAZOS, J. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, ACM, v. 47, n. 10, p. 147–162, 2012.

KUPERBERG, M.; KROGMANN, M.; REUSSNER, R. Timermeter: Quantifying properties of software timers for system analysis. In: *2009 Sixth International Conference on the Quantitative Evaluation of Systems*. [S.l.: s.n.], 2009. p. 85–94.

LAU, J. et al. Online performance auditing: Using hot optimizations without getting burned. *ACM SIGPLAN Notices*, ACM, v. 41, n. 6, p. 239–251, 2006.

LEATHER, H.; BONILLA, E.; O'BOYLE, M. Automatic feature generation for machine learning–based optimising compilation. *ACM Transactions on Architecture and Code Optimization*, ACM, New York, NY, USA, v. 11, n. 1, p. 14:1–14:32, fev. 2014. ISSN 1544-3566.

MAGNI, A.; DUBACH, C.; O'BOYLE, M. Automatic optimization of thread-coarsening for graphics processors. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. New York, NY, USA: ACM, 2014. (PACT '14), p. 455–466. ISBN 978-1-4503-2809-8.

MCGOVERN, A.; MOSS, E.; BARTO, A. G. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine learning*, Springer, v. 49, n. 2-3, p. 141–160, 2002.

MIGNON, A. S.; ROCHA, R. L. A. An application of composite nano-patterns to compiler selected profiling techniques. In: *Proceedings of the 6th International Conference on Software and Computer Applications*. New York, NY, USA: ACM, 2017. (ICSCA '17), p. 186–190. ISBN 978-1-4503-4857-7. Disponível em: ⟨http://doi.acm.org/10.1145/3056662.3056679⟩.

MITCHELL, T. M. *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997.

MONSIFROT, A.; BODIN, F.; QUINIOU, R. A machine learning approach to automatic production of compiler heuristics. In: SPRINGER-VERLAG. *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. London, UK, 2002. p. 41–50.

MURPHY, K. P. *Machine Learning: A Probabilistic Perspective*. [S.l.]: The MIT Press, 2012. ISBN 9780262018029.

OGILVIE, W. F. et al. Minimizing the cost of iterative compilation with active learning. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [S.l.: s.n.], 2017. p. 245–256.

ORACLE. *Java Agent API*. 2016. https://docs.oracle.com/javase/6/docs/api/. Acesso: Outubro de 2016.

_____. *JVM Tool Interface*. 2016. http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html. Acesso: Outubro de 2016.

PALLISTER, J.; HOLLIS, S. J.; BENNETT, J. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, v. 58, n. 1, p. 95–109, 2015.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, v. 12, p. 2825–2830, nov. 2011. ISSN 1532-4435.

SANCHEZ, R. N. et al. Using machines to learn method-specific compilation strategies. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2011. (CGO '11), p. 257–266. ISBN 978-1-61284-356-8.

SETTLES, B. *Active Learning*. [S.l.]: Morgan & Claypool Publishers, 2012. 1–114 p.

SHIV, K. et al. Specjvm2008 performance characterization. In: *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 17–35. ISBN 978-3-540-93798-2.

SINGER, J. et al. Fundamental nano-patterns to characterize and classify java methods. *Electron. Notes Theor. Comput. Sci.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 253, n. 7, p. 191–204, set. 2010. ISSN 1571-0661. Disponível em: ⟨http://dx.doi.org/10.1016/j.entcs.2010.08.042⟩.

STEPHENSON, M. et al. Meta optimization: Improving compiler heuristics with machine learning. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2003. (PLDI '03), p. 77–90. ISBN 1-58113-662-5.

SUTTON, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: *Advances in Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press, 1996. p. 1038–1044.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning: An Introduction*. 2nd (in progress). ed. Cambridge, MA, USA: MIT Press, 2017. Disponível em: ⟨http://incompleteideas.net/sutton/book/the-book-2nd.html⟩.