JORGE AUGUSTO SABALIAUSKAS

**AUTOMATIZAÇÃO DO PROCESSO DE SELEÇÃO DE
TRANSFORMAÇÕES PARA OTIMIZAÇÃO DO TEMPO
DE EXECUÇÃO POR MEIO DE APRENDIZADO DE
MÁQUINA NO ARCABOUÇO DA LLVM**

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia de Computação.

São Paulo
2015

JORGE AUGUSTO SABALIAUSKAS

**AUTOMATIZAÇÃO DO PROCESSO DE SELEÇÃO DE
TRANSFORMAÇÕES PARA OTIMIZAÇÃO DO TEMPO
DE EXECUÇÃO POR MEIO DE APRENDIZADO DE
MÁQUINA NO ARCABOUÇO DA LLVM**

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia de Computação.

Área de Concentração:

Sistemas Digitais

Orientador:

Ricardo Luis de Azevedo da Rocha

São Paulo
2015

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

São Paulo, 18 de maio de 2015.

Assinatura do autor

Assinatura do orientador

Catálogo-na-publicação

Sabaliauskas, Jorge Augusto

Automatização do Processo de Seleção de Transformações para Otimização do Tempo de Execução por Meio de Aprendizado de Máquina no Arcabouço da LLVM / J. A. Sabaliauskas. – ed. rev. –
– São Paulo, 2015.

72 p.

Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

1. Automatização do processo de ajuste de compiladores
2. Otimização de código 3. Ajuste de parâmetros de transformação
4. Aprendizado de máquina 5. LLVM - Low Level Virtual Machine
I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t. Departamento de Engenharia de Computação e Sistemas Digitais (PCS). II. t.

RESUMO

A rápida evolução do *hardware* demanda uma evolução contínua dos compiladores. Um processo de ajuste deve ser realizado pelos projetistas de compiladores para garantir que o código gerado pelo compilador mantenha uma determinada qualidade, seja em termos de tempo de processamento ou outra característica pré-definida. Este trabalho visou automatizar o processo de ajuste de compiladores por meio de técnicas de aprendizado de máquina. Como resultado os planos de compilação obtidos usando aprendizado de máquina com as características propostas produziram código para programas cujos valores para os tempos de execução se aproximaram daqueles seguindo o plano padrão utilizado pela LLVM.

ABSTRACT

The fast evolution of hardware demands a continue evolution of the compilers. Compiler designers must perform a tuning process to ensure that the code generated by the compiler maintain a certain quality, both in terms of processing time or another preset feature. This work aims to automate compiler adjustment process through machine learning techniques. As a result the compiler plans obtained using machine learning with the proposed features had produced code for programs whose values for the execution times approached those following the standard plan used by LLVM.

SUMÁRIO

Lista de Ilustrações

Lista de Tabelas

1	Introdução	11
1.1	Objetivo	12
1.2	Objetivos Específicos	13
1.3	Estado da Arte	14
1.4	Organização	16
2	Compiladores	17
2.1	Divisão Clássica	17
2.2	Representação Intermediária	20
2.3	Compilação Dinâmica	21
3	Aprendizado Supervisionado	23
4	Tratamento de dados	28
4.1	Boxplots, medianas e quartis	28
4.2	Principal Component Analysis	29
5	Metodologia	32

5.1	Seleção de Planos de Compilação	32
5.2	Vetor de Características	33
5.3	Processo de Aprendizado	33
6	Experimento	38
6.1	Arcabouço LLVM	39
6.2	Geração de Dados	41
6.3	Seleção de Exemplos	45
6.4	Aprendizado de Máquina	47
6.5	Programas de Treinamento e Teste	48
7	Resultados	52
7.1	CINT2000	53
7.2	CFP2000	53
7.3	Efeitos de Múltiplas Execuções	53
8	Análise dos Resultados	62
9	Conclusão	66
9.1	Trabalhos Futuros	68
	Referências	70

LISTA DE ILUSTRAÇÕES

1	Arquitetura clássica de um compilador	18
2	Exemplo de Aprendizado por Regressão Logística	26
3	Exemplo de Aprendizado por SVM	26
4	Exemplo de Árvore de Decisão	27
5	Exemplo genérico de boxplot	30
6	Exemplo de aplicação do PCA	31
7	Seleção dos melhores planos de compilação	33
8	Cálculo de Características	34
9	Construção de um exemplo para realização do treinamento	35
10	Aprendizado Supervisionado em Compiladores	36
11	Formação de um plano de compilação a partir de aprendizado de máquina	37
12	Exemplo de compilação usando a LLVM	40
13	Geração de dados de treinamento	44
14	Seletor de exemplos significativos	46
15	Tempo de execução para o programa gap	54
16	Tempo de execução para o programa twolf	54
17	Tempo de execução para o programa vpr	55
18	Tempo de execução para o programa gcc	55

19	Tempo de execução para o programa bzip2	56
20	Tempo de execução para o programa wupwise	56
21	Tempo de execução para o programa mesa	57
22	Tempo de execução para o programa equake	57
23	Tempo de execução para o programa fma3d	58
24	Tempo de execução para o programa facerec	58
25	Tempo de execução para o programa sixtrack	59
26	Tempo de execução para o programa mgrid	59
27	Tempo de execução mediano para três execuções do experimento no <i>cluster</i>	60
28	Tempo de execução mediano para três execuções do experimento no PC	61
29	Mediana do tempo de execução do programa sixtrack executado 100 vezes em 5 momentos distintos.	61

LISTA DE TABELAS

1	Tranformações alvo de aprendizado de máquina	41
2	Vetor de Características Utilizado	48
3	Parâmetros dos Algoritmos de Aprendizado	49
4	Programas do CINT2000	50
5	Programas do CFP2000	51
6	Especificação dos Ambientes de Execução	53
7	Resultado das otimizações sobre os programas CINT	63
8	Resultado das otimizações sobre os programas CFP	63
9	Análise dos dados de treinamento	65
10	Análise do Vetor de Características	65

1 INTRODUÇÃO

Compiladores podem ser considerados a ponte entre o *hardware* e *software*, seu desenvolvimento deve ser tanto capaz de extrair o potencial de novos *hardwares* quanto atender às necessidades das novas linguagens de programação, sendo tão importante quanto os semicondutores na era da informação [12].

O rápido avanço dos *hardwares* exige alterações frequentes nos compiladores. Essa mudança acaba sendo lenta devido a um ajuste necessário para cada arquitetura suportada pelo compilador [25, 22, 28, 9]. O ajuste tem como objetivo configurar o compilador para que o código produzido possua determinado nível de qualidade em cada arquitetura.

Cada arquitetura computacional necessita de tratamento especial por parte do compilador [2, 10, 30, 21]. A geração de código é particular para cada arquitetura, transformações que produzem um código eficiente em uma arquitetura não necessariamente produzem o mesmo efeito em outra. Para cada arquitetura é necessário um ajuste específico.

O ajuste, geralmente, é feito de maneira manual por um especialista, ficando ele responsável por configurar parâmetros que afetam a eficiência do programa gerado [25, 28]. Esses parâmetros são configurados segundo heurísticas, isto é, valores experimentais para parâmetros que sejam capazes de produzir bons programas (não necessariamente asseguram o melhor resultado) [2]. Essas heurísticas não garantem resultados ótimos e são dependentes de vários fatores como arqui-

tetura, tipo de aplicação entre outros.

Este trabalho analisa o potencial do aprendizado de máquina para a elaboração do plano de compilação (conjunto de transformações aplicadas sobre um programa) presente em um compilador *com o objetivo de diminuição no tempo de execução*. Para isto foi construído um experimento sobre o arcabouço LLVM [15] no qual o processo de habilitação e desabilitação de algumas transformações foi efetivado através de aprendizado de máquina em vez de utilizar heurísticas buscando seguir os trabalhos de [22, 9, 25].

Uma exploração dos possíveis planos de compilação foi feita com o objetivo de gerar dados para os algoritmos de aprendizado. Esses dados eram a associação de uma função do programa a um conjunto de transformações necessárias para o compilador gerar um programa mais eficiente. Nesse caso um bom programa é aquele que apresenta um menor tempo de execução em relação ao mesmo programa compilado sem otimização.

A montagem do experimento possibilitou a comparação de vários algoritmos de aprendizado de máquina. Nesse trabalho foram estudados os algoritmos de regressão logística, *Support Vector Machines* (SVM) e árvores de decisão.

Para critério de avaliação da otimização realizada pelo experimento foram utilizados programas do *benchmark* SPEC cpu2000 [14] para processadores.

1.1 Objetivo

Esta pesquisa tem por objetivo principal estruturar um procedimento para automatizar o processo de ajuste de compiladores por meio de técnicas de aprendizado de máquina. A infraestrutura básica empregada foi o arcabouço da LLVM.

Para realizar o objetivo faz-se necessária a análise dos métodos de aprendizado de máquina para a elaboração de planos de compilação a partir de exemplos.

Esses exemplos são outros planos de compilação capazes de reduzir o tempo de execução de programas. Os exemplos são produzidos através de uma busca simples no espaço de planos de compilação.

A viabilidade de tal processo permite a semi-automatização do processo de ajuste de um compilador, não é uma automatização absoluta já que o processo depende de um disparo manual do processo de aprendizado.

1.2 Objetivos Específicos

Como objetivos específicos tem-se:

1. Análise de métodos de aprendizado de máquina: Nesta pesquisa os métodos investigados foram Regressão Logística, SVM e Árvores de Decisão;
2. Exploração das transformações da LLVM: Os programas são divididos em um conjunto de funções, assim as transformações que atuavam no contexto de funções fornecidas pela própria LLVM foram estudadas e utilizadas.

Sabendo-se que um programa pode ser dividido em um conjunto de funções, gerou-se para cada função um plano de compilação. Assim, empregando um método de aprendizado de máquina produz-se um modelo e se utiliza esse modelo aprendido para o processo de ajuste. Essa é a mesma idéia de geração de planos individuais usada em [9, 25].

Para a construção do otimizador foi utilizado o arcabouço LLVM. Como transformações visando otimizar código, dentro do contexto das funções, foram utilizadas transformações fornecidas pela própria LLVM.

Os algoritmos de aprendizado de regressão logística, SVM e árvores de decisão, já usadas nos trabalhos de [9], [25] e [22] respectivamente, foram aplicados agora no contexto da LLVM.

1.3 Estado da Arte

O aprendizado de máquina em compiladores foi aplicado inicialmente com o objetivo de automatizar o processo de geração de heurísticas [9, 25], processo que é realizado manualmente.

Heurísticas são utilizadas no processo de otimização por consumirem menor tempo de compilação em relação a técnicas mais sofisticadas de otimização discreta geralmente não produzindo resultados ótimos [2].

O processo de produção de uma heurística consiste em ajustar parâmetros de um compilador até que determinada qualidade de código seja obtida [28]. Para se chegar em um compilador a nível de produção podem ser necessários anos de ajuste [25].

Uma das primeiras aplicações do aprendizado de máquina em compiladores foi na transformação *looping unrolling* por [22]. Essa transformação foi escolhida por ser fácil de implementar (basta replicar o corpo de um laço) e possuir muitos efeitos colaterais (replicar indiscriminadamente o laço causa um efeito negativo na execução de um programa). O uso de árvores de decisão foi escolhido como método de aprendizagem por permitir a compreensão do modelo gerado por um ser humano.

O estudo sobre a transformação *loop unrolling*, replicação do corpo de um laço um determinado número de vezes para se tirar proveito do paralelismo a nível de instruções em processadores, foi feito utilizando o compilador GNU Fortran alterando-se a heurística de ativação de *loop unrolling* para utilizar um modelo aprendido a partir de exemplos. O algoritmo de aprendizado utilizado era baseado em árvores de decisão cujo modelo gerado poderia ser interpretado por um especialista.

Baseando-se na ideia de que para cada transformação pode-se associar uma

prioridade (custo da transformação), [28] aplicou programação genética para gerar funções de prioridade para três transformações: hyper-blocos, alocação de registradores e leitura antecipada. O cálculo das funções de prioridade passou a ser feito por um par interpretador-programa. Resultados mostraram pequenas melhorias em todos os *benchmarks*.

O trabalho de [9] aplicou a técnica de *logistic regression* para habilitar ou desabilitar transformações na JikesRVM ¹. Nesse trabalho foi percebido que as transformações poderiam obter melhores resultados em termos de otimização se fossem selecionadas individualmente para cada método. A JikesRVM foi modificada para compilar métodos individuais variando-se transformações aleatoriamente, medição de tempo foi adicionada a cada método, as transformações com melhor tempo de execução foram selecionadas e usadas para o treinamento da *logistic regression*. O mecanismo de decisão usando aprendizado de máquina superou a configuração padrão presente na JikesRVM.

Com o início das aplicações do aprendizado de máquina na área de compiladores [17] argumentou que apenas o trabalho de busca de heurísticas tinha sido substituído pelo trabalho de busca de características. O trabalho de [17] propôs um método para geração de características utilizando programação genética. O cálculo de componentes do vetor de característica passou a ser feito por um pequeno programa que era gerado usando programação genética [20]. Componentes do vetor de características eram adicionados somente se, após o aprendizado, fosse possível notar uma melhora no modelo produzido pelo aprendizado de máquina. Esse método foi capaz de superar os vetores de características existentes.

O trabalho de [25] avançou em direção ao uso do aprendizado supervisionado aplicando-o em uma máquina virtual de grande porte: a Testarossa da IBM. Nesse caso foi utilizada Support Vector Machine (SVM) como mecanismo de trei-

¹Máquina virtual Java implementada em Java para fins acadêmicos

namento. A mudança nos planos de compilação tinha de ser feita cuidadosamente pois podia-se obter um trecho de programa defeituoso caso a ordem de transformações fosse alterada. No mecanismo de treinamento era permitido apenas a ativação ou não dos planos de compilação. O processo de modificação de planos partia do plano original da Testarossa fazendo pequenas alterações ao longo do tempo em busca de melhorias locais. O desempenho de não sofreu alteração mas foi possível melhorar o tempo de inicialização.

1.4 Organização

Na primeira parte desse trabalho é apresentada a teoria básica de compiladores e aprendizado de máquina junto com alguns métodos para análise de dados. Na segunda parte é apresentado como foi elaborado o experimento, os resultados obtidos a partir dele e sua análise para uma posterior conclusão.

2 COMPILADORES

Nesse capítulo é apresentada a arquitetura de um compilador. É apresentada inicialmente a divisão clássica descrita em documentos técnicos da área [2, 10] e, ao final, uma breve descrição sobre a representação intermediária.

2.1 Divisão Clássica

Compiladores transformam uma linguagem de entrada em uma linguagem de saída. Geralmente a linguagem de entrada é um texto representando um programa e a linguagem de saída é uma linguagem de máquina compreendida diretamente pelo computador.

Um compilador é organizado por uma série de passos onde a saída de um passo é consumida pelo passo seguinte. A entrada do primeiro passo é o código-fonte do programa e a saída do último passo é o código de máquina. Alguns passos geralmente presentes são o analisador léxico, analisador sintático, otimizador independente de máquina, seletor de instruções, alocador de registradores e agendador de instruções. Uma visão geral da arquitetura é descrita na Figura 1.

O analisador léxico é responsável por identificar as palavras presentes no código-fonte pertencentes à linguagem de programação. Essa identificação é feita através do agrupamento de sequência de caracteres pertencentes a uma mesma palavra e posterior classificação. A saída do analisador léxico é um par indicando

Figura 1: Arquitetura clássica de um compilador. A saída de cada passo é utilizada pelo passo seguinte. As exceções são para o primeiro passo onde a entrada é o código fonte e para o último passo onde a saída é o código de máquina.

a classe e o texto da palavra. Esse par é conhecido como *token*.

O analisador sintático tem como objetivo verificar se a disposição das palavras obedece a gramática definida pela linguagem de programação. Essa estrutura é definida através de uma gramática que indica por meio de regras como produzir o texto de um programa correto. Fica a cargo do analisador sintático determinar quais regras da gramática foram utilizadas para produzir o código fonte. Caso ele não consiga encontrar essas regras é porque o programa não está de acordo com a estrutura da linguagem e a compilação é abortada.

O analisador sintático geralmente faz a verificação apenas da estrutura da linguagem, não detectando erros relacionados a linguagem de programação como, por exemplo, o uso de variável não declarada. A detecção desses erros fica sob responsabilidade do verificador de tipos, que utilizará mais informações sobre a semântica (significado) da linguagem do que o analisador sintático.

O otimizador independente de máquina tem como objetivo melhorar o código gerado pelo compilador. Apesar de chamado de otimização ele não encontra um programa ótimo mas sim um programa equivalente melhor com relação ao tempo de execução, tamanho de código ou consumo de energia. Esse programa equivalente melhor é encontrado através da aplicação de transformações sobre o programa original.

As transformações que serão aplicadas pelo otimizador independente de máquina estão descritas em um plano de compilação. Esse plano de compilação é elaborado através da identificação de características do programa que podem ser melhoradas através de certas transformações.

O seletor de instruções procura pela melhor instrução de máquina capaz de executar determinada operação do programa. A partir desse passo é necessário um conhecimento profundo sobre a arquitetura alvo do programa executável.

Na execução de um programa é comum a necessidade de se lidar com diversos valores temporários. Esses temporários são armazenados em uma memória interna do processador chamada de registradores. A quantidade de dados total que pode ser armazenada em registradores é pequena, porém seu acesso é extremamente rápido. Fica sob a responsabilidade do alocador de registradores determinar quais temporários devem ser armazenados em registradores.

Por último, o agendador de instruções tem como objetivo determinar a ordem das instruções. Devido a capacidade de um processador moderno executar mais de uma instrução ao mesmo tempo, o agendador de instruções deve tentar organizar as instruções para tirar o maior proveito desse paralelismo.

Ainda na Figura 1, é possível verificar que certos componentes podem ser classificados em *frontend*, otimizador e *backend*. Os passos do *frontend* ficam responsáveis pelo processamento da linguagem de programação enquanto que os passos do *backend* ficam responsáveis pela geração de código de máquina. Essa divisão é feita para indicar que um mesmo *backend* pode ser utilizado por muitos *frontends*, isto é, diferentes linguagens de programação.

2.2 Representação Intermediária

Cada forma de representação do programa dentro do compilador recebe o nome de Representação Intermediária [27]. Em cada representação intermediária pode-se variar a quantidade de informação retida do programa fonte, o tipo de representação e a política de geração de símbolos [10].

A quantidade de informação retida do programa fonte em uma representação intermediária tende a diminuir até a geração de código. Transformações e análises do *frontend* tendem a trabalhar no nível do código fonte enquanto que as transformações e análises no *backend* tendem a trabalhar no nível da linguagem

de máquina.

Diferentes tipos de representação intermediária podem ser empregadas conforme a necessidade. Entre os possíveis tipos de representações intermediárias temos a representação em grafo e a representação linear.

2.3 Compilação Dinâmica

Um tipo especializado de compilador pode ser encontrado em máquinas virtuais ¹ para geração de código em tempo de execução. Este tipo de compilador é chamado de *Just-in-time* JIT e é utilizado para melhoria de execução de um programa [4] durante o tempo de execução.

Devido a um compilador JIT poder atuar durante a execução de um programa, ele é capaz de direcionar melhor a geração de código devido a informações presentes em tempo de execução [4]. Devido a competição com o programa por recursos, o compilador dinâmico deve ser projetado cuidadosamente para não interferir no programa em execução.

Através da coleta de informações em tempo de execução pode-se selecionar um trecho do programa em execução para ser submetido ao compilador dinâmico, processo chamado por [3] de otimização seletiva. Este código pode ser compilado várias vezes usando cada vez um número maior de otimizações (para minimizar o tempo de compilação) e o código gerado descartado quando desnecessário [4].

Uma forma de aplicar os benefícios da compilação dinâmica em compiladores tradicionais é através da coleta dos dados de tempo de execução de um programa (frequência de execução de caminhos, tempos de execução, informações fornecidas pelo *hardware* entre outras). Esses dados de execução são posteriormente

¹Apesar do termo "máquinas virtuais" também ser utilizado para denominar *softwares* capazes de emular um sistema computacional completo, restringir-se-á nesse trabalho a máquinas virtuais destinadas unicamente a execução de programas

fornecidos ao compilador para auxiliar o processo de compilação. Esse processo é chamado de *Profile Guided Optimization* [26].

3 APRENDIZADO SUPERVISIONADO

O aprendizado de máquina busca fazer com que computadores passem a aprender de maneira parecida com o ser humano. Algoritmos de aprendizado de máquina são divididos conforme o tipo de aprendizado. Os tipos de aprendizado de máquina são o supervisionado, o não-supervisionado e o por reforço. Essa trabalho irá utilizar apenas o aprendizado supervisionado, informações sobre o aprendizado não-supervisionado e por reforço podem ser vistos em [20].

O aprendizado supervisionado consiste em fazer o computador aprender uma função a partir de exemplos [20, 11, 7, 23]. Vamos supor a predição do preço de um imóvel dada uma região. Como exemplos teríamos associações de região com preço de imóvel. Um algoritmo de aprendizado supervisionado processaria esses exemplos e produziria um modelo capaz de prever o preço de um imóvel para qualquer região, inclusive regiões não contempladas nos exemplos. O processo de aplicação de um algoritmo de aprendizado supervisionado sobre os exemplos é chamado de treinamento.

A predição do preço para uma região não contemplada é feita através de uma generalização dos exemplos usados no treinamento [7]. Como a forma em que o algoritmo de aprendizado supervisionado generaliza o conjunto de exemplos para produzir o modelo varia de algoritmo para algoritmo, é necessário avaliar os diferentes algoritmos na predição de valores não vistos.

De maneira mais formal, no aprendizado supervisionado tenta-se encontrar

uma função $h(X)$ a partir de exemplos $\{(X^{(1)}, Y^{(1)}), \dots, (X^{(n)}, Y^{(n)})\}$ ($X^{(i)}$ representa a entrada do i -ésimo exemplo e $Y^{(i)}$ representa a saída do i -ésimo exemplo) tal que $h(X^{(i)}) = Y^{(i)}$ com $i \in \{1, \dots, n\}$. A função $h(X)$ é chamada historicamente de hipótese [20], representaria o modelo aprendido pelo computador para representar o conceito expresso a partir dos exemplos.

Podemos dividir o aprendizado supervisionado a partir do tipo de saída (Y) em:

Classificação Quando os possíveis valores de Y são valores discretos. Costuma-se chamar cada valor de Y de classe ou rótulo. Como exemplo temos $Y \in \{0, 1\}$.

Regressão Quando os possíveis valores de Y são valores contínuos. Como exemplo podemos aprender o preço do valor de um imóvel para uma região.

O tipo de aprendizado usado nesse trabalho será para a classificação de valores, mais especificamente, apenas para prever as saídas $Y \in \{0, 1\}$.

Nesse trabalho os seguintes algoritmos de classificação serão utilizados:

Regressão Logística É um algoritmo de classificação apesar do nome. Esse algoritmo tenta separar os exemplos em 0 e 1 usando a combinação de um hiperplano com a função logística: $f(x) = \frac{1}{1+e^x}$. Um exemplo de classificação usando regressão logística pode ser visto na Figura 2. A regressão logística foi usada no trabalho [9] para a habilitação de transformações em planos de compilação. Maiores detalhes sobre regressão logística pode ser vista em [7, 23, 13].

Support Vector Machine Máquinas de vetor de suporte buscam separar os exemplos através da maximização de uma margem entre os exemplos mais próximos entre cada classe. Para exemplos não-linearmente separáveis

pode-se aplicar uma transformação usando uma função *kernel* para um espaço de maior dimensão onde os exemplos sejam linearmente separáveis. Um exemplo de classificação usando SVM pode ser visto na Figura 3. O trabalho [25] utilizou SVM para a habilitação de transformações em planos de compilação. Para maiores detalhes sobre SVM consultar [13, 7, 23].

Árvores de Decisão Árvores de decisão buscam dividir o problema de classificação em subproblemas através de um atributo que melhor divide os dados de treinamento. Ao não ser mais possível dividir os dados de treinamento então gera-se uma folha e associa-se a saída Y que ela melhor representa. Um exemplo de árvore pode ser vista na Figura 4. Os fundamentos teóricos de árvores de decisão podem ser vistos em [20, 7, 23]. Árvores de decisão foram usadas em [22] na configuração de parâmetros da transformação *loop unrolling*.

Um problema que se pode encontrar ao realizar aprendizado de máquina é que o modelo produzido pode ser capaz de prever com grande exatidão os dados usados no treinamento mas apresenta uma grande taxa de erros ao ser apresentado a dados novos [7, 23, 11, 20]. Um método comum utilizado para avaliar a capacidade de generalização de um modelo é dividir os exemplos em exemplos de treinamento e exemplos de teste [20]. Os exemplos de treinamento são utilizados para a produção do modelo de predição enquanto que os exemplos de teste são usados para avaliar a capacidade de generalização do modelo de predição. Esse método é chamado de validação cruzada.

Figura 2: Exemplo de Aprendizado por Regressão Logística. A reta representa a superfície de decisão que melhor separa os exemplos.

Figura 3: Exemplo de Aprendizado por SVM. A reta representa a superfície de decisão (usando um *kernel* linear) que melhor separa os exemplos.

Figura 4: Exemplo de uma Árvore de Decisão. Cada nó representa uma decisão sobre um componente do vetor de características e cada folha é associada a um valor. Do vetor de características estimar-se-á um valor percorrendo um caminho do nó raiz à uma folha realizando decisões sobre componentes do vetor de características.

4 TRATAMENTO DE DADOS

Nesse capítulo são mostrados os métodos de tratamento de dados utilizados. Uma revisão básica sobre *boxplots*, medianas e quartis é apresentada na seção 4.1. Na seção 4.2 é mostrado o método PCA usado geralmente para analisar os dados para aprendizado de máquina.

4.1 Boxplots, medianas e quartis

Medianas e quartis são medidas-resumo calculados a partir da ordenação dos dados de um experimento que apresentam pouca influência na presença de dados discrepantes [8]. Para ilustrar o cálculo dessas medidas-resumo, suponha que um experimento qualquer foi realizado e os seguintes dados obtidos:

3, 5, 4, 8, 2, 5, 6, 8, 4, 3

Ordenando os valores obtemos:

2, 3, 3, 4, 4, 5, 5, 6, 8, 8

A mediana (m) é definida como o número que se encontra no meio da sequência dos dados ordenados. Caso a sequência apresente um número par de elementos não haverá um elemento no meio da sequência, porém, pode-se considerar como mediana a média aritmética dos dois valores que se encontram no meio da sequência. Para o caso da sequência acima a mediana é dada por:

$$m = \frac{4+5}{2} = 4.5$$

Os quartis são obtidos de forma similar à mediana. Considerando a sequência s_1 como os $n/2$ valores menores que m e s_2 como os $n/2$ valores maiores que m , então definimos os quartis Q_1 e Q_3 como as medianas de s_1 e s_2 respectivamente. Para os dados acima podemos extrair os seguintes valores para Q_1 e Q_3 :

$$s_1 = 2, 3, 3, 4, 4 \implies Q_1 = 3$$

$$s_2 = 5, 5, 6, 8, 8 \implies Q_3 = 6$$

Dados os valores da mediana, e dos quartis Q_1 e Q_3 , podemos representar graficamente esses valores através de um gráfico chamado de *boxplot* (Figura 5). No *boxplot* é possível visualizar o espalhamento de dados, os valores da mediana, os quartis, pontos extremos (menor e maior valores) e possíveis pontos discrepantes. Para a identificação de pontos discrepantes no *boxplot* os seguintes valores são calculados:

$$LI = Q_1 - 1,5 * (Q_3 - Q_1)$$

$$LS = Q_3 + 1,5 * (Q_3 - Q_1)$$

Valores que são maiores ou menores do que o limite superior (LS) e inferior (LI) respectivamente são considerados como discrepantes. Na Figura 5 os valores discrepantes são representados por uma cruz acima ou abaixo dos limites inferiores e superiores.

Para maiores detalhes sobre medianas, quartis e *boxplots* consultar [8].

4.2 Principal Component Analysis

Principal Component Analysis (PCA) é uma técnica que usa uma transformação linear para produzir um novo sistema de coordenadas onde a variância seja

Figura 5: Exemplo genérico de boxplot. No *boxplot* estão representadas a mediana (m), o primeiro quartil ($Q1$), o terceiro quartil ($Q3$), o limite inferior (LI) e o limite superior (LS).

maximizada ao longo de cada eixo e a covariância anulada entre eixos distintos. Na Figura 6 é possível ver um exemplo de aplicação de PCA.

Ao aplicar o PCA é possível obter uma porcentagem do total de variância para cada um dos componentes. Removendo-se os componentes com menor porcentagem do total de variância é uma forma de reduzir o número de componentes perdendo o mínimo de informação.

Devido ao critério de seleção de componentes segundo a variância, o PCA pode ser aplicado para reduzir o tamanho do vetor de características em aprendizado de máquina para eliminar características pouco significativas. Antes de aplicar algum algoritmo de aprendizado pode-se aplicar o PCA sobre os dados disponíveis, selecionar os componentes com maior percentual de variância, determinar a transformação linear, aplicar a transformação sobre os dados de aprendizado e, por fim, aplicar o algoritmo de aprendizado. Toda vez que se desejar fazer uma predição, deverá ser transformado o vetor de características para as novas coordenadas usando a transformação linear, remover os componentes pouco significativos e, por fim, realizar a predição usando o algoritmo de aprendizado sobre o vetor de características transformado.

(a) Antes do PCA

(b) Depois do PCA

Figura 6: Exemplo de aplicação do PCA. Um novo sistema de coordenadas (w_1 e w_2) é produzido de forma a maximizar a variância em torno de cada eixo e anular as covariâncias entre cada um dos componentes.

5 METODOLOGIA

A partir dos trabalhos feitos [22, 9, 25] é possível identificar um processo para a aplicação de aprendizado de máquina supervisionado em compiladores. Essa metodologia pode ser aplicada tanto aos compiladores tradicionais quanto aos específicos presentes em máquinas virtuais. Este trabalho focará apenas nos compiladores tradicionais.

No aprendizado supervisionado deseja-se que o computador seja capaz de relacionar uma entrada a uma saída. Em compiladores a saída que deseja-se prever é um plano de compilação a partir de uma entrada calculada sobre o programa. Nas próximas seções será discutido como se obter os planos de compilação, como pode ser calculada o vetor de características sobre o programa e por último a visão geral de como realizar o aprendizado supervisionado.

5.1 Seleção de Planos de Compilação

O plano de compilação é o responsável por guiar as transformações aplicadas sobre o programa (ou parte dele) a fim de melhorar determinada característica. Para realizar o aprendizado supervisionado será necessário saber quais planos de compilação deverão ser produzidos dependendo do programa.

A Figura 7 fornece um método para se obter planos de compilação. O resultado da compilação de um programa fixo é avaliado para cada plano de compilação segundo uma característica desejada (tempo de execução, tamanho de código,

Figura 7: Seleção dos melhores planos de compilação. É selecionado o plano de compilação que produza o melhor código para um programa fixado.

gasto de energia, etc.). O melhor plano é selecionado para ser relacionado com o programa de entrada, nesse caso o programa fixo.

5.2 Vetor de Características

O vetor de características tentará expressar características do programa de entrada correlacionadas com o plano de compilação, isto é, cada componente do vetor de características deverá influenciar na decisão do plano de compilação.

Na Figura 8 é possível ver um método para o cálculo do vetor de características. Os componentes do vetor podem ser calculados através da contagem ou da verificação da presença de certas instruções. Em ambos os diferentes tipos de cálculo de componentes é necessário o conhecimento sobre o funcionamento do compilador para saber se é possível obter algum benefício através de alguma transformação presente no plano de compilação.

5.3 Processo de Aprendizado

Com os planos de compilação desejados e um vetor de características capaz de representar um programa, a última etapa consiste em preparar os dados de

Figura 8: Cálculo de Características. Em a) é mostrado um fragmento do código fonte de um programa qualquer, em b) é mostrada a representação intermediária de árvore para esse fragmento de código e em c) é mostrado o vetor de características calculado sobre a representação intermediária.

treinamento para o aprendizado supervisionado.

O plano de compilação pode ser considerado como um vetor binário, onde cada componente representa uma transformação que será aplicada pelo compilador. Quando um componente i do vetor de características recebe o valor 1 significa que a transformação i está habilitada, 0 significa que a transformação i está desabilitada.

Na Figura 9 pode-se ver a construção de um exemplo para o treinamento. Cada exemplo para treinamento relaciona o vetor de características com uma transformação do plano de compilação. Esse plano de compilação é aquele que apresenta a característica desejada para o código obtido pelo compilador.

Dado que os valores possíveis da predição da habilitação de uma transformação pertence ao conjunto $\{0, 1\}$, então o problema de aprendizado é do tipo classificação. Será necessário um classificador para cada transformação descrita

Figura 9: Construção de um exemplo para realização do treinamento. Um exemplo para o treinamento é obtido pelo mapeamento dos características calculados sobre a representação intermediária com uma transformação do plano de compilação. Será necessário um modelo de predição para cada componente do plano de compilação.

Figura 10: Aprendizado Supervisionado em Compiladores. $X^{(1)}, \dots, X^{(n)}$ exemplos serão usados para produzir um modelo $h(X)$ capaz de prever a habilitação de uma transformação dados características calculados sobre a representação intermediária do programa.

no plano de compilação.

Criando um conjunto de treinamento com os exemplos de como habilitar transformações segundo o programa de entrada, basta submeter este conjunto de treinamento a um algoritmo de aprendizado supervisionado. O algoritmo de aprendizado supervisionado fornecerá um modelo capaz de prever a habilitação de cada transformação em um plano de compilação.

Na Figura 11 é possível ver a esquematização da construção de um plano de compilação para um programa. Dado o programa representado na representação intermediária, seu vetor de características é calculado e submetido aos modelos produzidos pelo algoritmo de aprendizado supervisionado que indicarão a habilitação das t transformações presentes no plano de compilação.

Figura 11: Formação de um plano de compilação a partir de aprendizado de máquina. Em a) é possível ver a representação intermediária de um programa, b) representa o vetor de características calculado sobre a representação intermediária, c) são os modelos para habilitação de transformações obtidos através de aprendizado de máquina e d) representa o plano de compilação com t transformações.

6 EXPERIMENTO

O experimento pode ser dividido em três fases: a primeira gera exemplos de compilação a partir de um conjunto de programas, a segunda treina um processo de elaboração de planos de compilação usando aprendizado de máquina e a terceira aplica o processo de elaboração em outro conjunto de programas para avaliação.

Na geração de dados é feita uma busca no espaço de planos de compilação usando um conjunto programas a fim de encontrar bons planos de compilação para serem utilizados como exemplos no treinamento dos algoritmos de aprendizado. Bons planos de compilação, nesse caso, são aqueles que quando aplicados pelo compilador produzem um programa que executa em menor tempo. O conjunto de programas usado nessa fase são identificados como programas de treino.

Os exemplos são usados nos algoritmos de aprendizado para a produção do processo de elaboração de planos de compilação. Nos exemplos deverão estar presentes tanto exemplos de ativação quanto de desativação de transformações. Os modelos de classificação por regressão logística, SVM e árvore de decisão são aplicados nessa fase.

Para avaliar o processo de elaboração de planos de compilação usando aprendizado de máquina é utilizado um conjunto de programas diferentes dos usados na etapa de geração de dados chamados de programas de teste. No final foram produzidos gráficos comparativos entre os algoritmos de aprendizado de máquina

e ambientes de execução usados pelo experimento.

A partir do gráfico comparativo era possível responder questões de que se o plano de compilação elaborado usando aprendizado de máquina era competitivo com o plano de compilação padrão e qual era o melhor algoritmo de aprendizado de máquina.

O arcabouço escolhido para uso nesta pesquisa foi o LLVM – *Low Level Virtual Machine* [15], que é composto de códigos fonte e de documentos abertos e livres. Além dessa característica desejável, o arcabouço é utilizado amplamente tanto na academia [6, 5] quanto na indústria [18, 31, 29].

Nas próximas seções são tratados detalhes do arcabouço LLVM, do processo de geração de dados, como foram selecionados exemplos significativos, como foi feito a aplicação do aprendizado de máquina e detalhes sobre os programas de treinamento e de teste.

6.1 Arcabouço LLVM

Para a construção do otimizador aproveitou-se do fato da LLVM já possuir implementada diversas transformações além de possuir suporte a diversos *frontends* (implementados exclusivamente sobre a LLVM ou *plugins* para compiladores existentes) [15, 1].

A LLVM permite a criação de transformações sobre programas através da implementação de passos. Passos permitem manipular a representação intermediária do programa e, opcionalmente, indicar dependências de outros passos. Passos podem operar a nível de laços, funções, grafo de chamada e módulos (conjunto de funções).

Os passos utilizados pelo aprendizado de máquina nesse trabalho são os do tipo que operam sobre funções. Para cada função do programa será elaborado

Figura 12: Exemplo de compilação usando a LLVM. Os componentes em cor escura são componentes externos, alterações são feitas apenas sobre o otimizador através da criação de passos adicionais sobre a LLVM.

um plano de compilação específico onde as transformações são habilitadas usando aprendizado de máquina.

A LLVM armazena a representação intermediária de um programa em um arquivo chamado de *bitcode* [16]. Um arquivo no formato de *bitcode* possui, geralmente, a extensão ".bc" (sufixo do nome do arquivo).

O esquema geral de compilação usado pela LLVM é descrito na Figura 12. Nessa figura estão presentes três componentes: um compilador, um otimizador e um ligador. O compilador representa o *frontend* que traduz o programa (I) para a representação intermediária que é armazenada em um arquivo *bitcode* (II). O otimizador realiza transformações sobre a representação intermediária e a armazena em um novo arquivo de *bitcode* (III). O ligador transforma a representação intermediária em um programa executável (IV). Os componentes representados por cores escuras são externos, alguns são a combinação de dois ou mais programas como o caso do ligador (aqui foi chamado de ligador apenas para simplificar a função da combinação de vários programas).

Os dados trocados na Figura 12, no escopo desse projeto, foram:

- I Arquivos de código fonte em C, C++ ou Fortran representados pelas extensões ".c", ".cpp" e ".f" respectivamente.
- II *Bitcode* gerado pelo compilador representado pela extensão ".bc". Para um mesmo programa podem ser gerados vários arquivos de *bitcode*, cada um representando uma transformação diferente.
- III *Bitcode* otimizado representado pela extensão ".bc".

Transformação	Descrição
Combine redundant instructions	Combina instruções realizando simplificação algébrica
Value Propagation	Propagação de valores
Dead Store Elimination	Eliminação de instruções de escrita redundantes
Simplify the CFG	Eliminação de códigos inatingíveis, fusões de blocos básicos e algumas transformações peephole
MemCpy otimization	Eliminação ou inserção de instruções <i>memcpy</i>
Aggressive Dead Code Elimination	Eliminação de código inatingível

Tabela 1: Transformações alvo de aprendizado de máquina.

IV Arquivo executável representado pela extensão ".exe". Foi utilizada essa extensão apenas como efeito de ilustração já que alguns sistemas operacionais não exigem uma extensão específica para programas executáveis.

Dentre as transformações disponíveis na LLVM, apenas as da Tabela 1 foram utilizadas no plano de compilação usando aprendizado de máquina. Essas transformações (junto com outras não necessariamente operando sobre funções) são inseridas pelo plano de compilação O1, um plano de compilação padrão presente na própria LLVM (de O1, O2 e O3 representam planos de compilação do menor para o maior número de transformações habilitadas sobre um programa).

6.2 Geração de Dados

Para determinar se uma transformação da Tabela 1 deve ser ativada sobre uma função precisa-se saber o efeito que ela produz no tempo de execução do programa (aumenta ou diminui). A estimativa desse efeito foi realizada de forma aproximada usando aprendizado de máquina através de um conjunto de exemplos relacionando funções do programa com o efeito de ativação. O algoritmo de geração de dados pode ser visto na Figura 1.

Entrada: P, τ, N

Saída: Exemplos de treinamento para as transformações $\tau_1.. \tau_K$

início

```

para cada  $P$  pertencente ao conjunto de treinamento faça
  para  $i \in \{1..N\}$  faça
    Seja  $\Delta t_i$  a medida do tempo de execução de  $P$  com  $\tau_1.. \tau_K$ 
    desabilitadas
  fim
  para cada transformação  $\tau \in \{\tau_1.. \tau_K\}$  faça
    para cada função  $f$  do programa  $P$  faça
      para  $i \in \{1..N\}$  faça
        Seja  $t_i(P, f, \tau)$  o tempo de execução de  $P$  com  $\tau$ 
        habilitada apenas sobre  $f$ 
        se  $t_i(P, f, \tau)$  e  $\Delta_i, i \in \{1..N\}$ , são diferentes então
          Calcular  $y \in \{0, 1\}$  usando  $t_i(P, f, \tau), \Delta_i, i \in \{1..N\}$ 
          Adicionar  $(x(f), y)$  aos dados de treinamento de  $\tau$ 
        fim
      fim
    fim
  fim
fim
fim

```

Algoritmo 1: Algoritmo de Geração de Dados. O algoritmo encontra casos em que a habilitação ou desabilitação de uma transformação sobre uma função de um programa de treino produza algum efeito no tempo de execução em relação à esse mesmo programa com as todas as transformações sobre função desabilitadas.

Nos exemplos, para cada função foram associados os seguintes rótulos:

- 0 Aumento no tempo de execução, não é vantajoso aplicar a transformação.
- 1 Diminuição no tempo de execução, é vantajoso aplicar a transformação.

No caso de um exemplo onde uma função associada a uma transformação acarreta em uma diminuição de tempo na execução de um programa então, para funções parecidas, deseja-se que a mesma transformação seja habilitada no plano de compilação, caso contrário, se essa função apresenta um aumento no tempo da execução, para funções parecidas, deseja-se que essa transformação esteja desabilitada no plano de compilação.

O esquema para a produção dos exemplos de compilação (dados de treinamento) feito a partir do algoritmo da Figura 1 pode ser visto na Figura 13. Nessa figura, as seguintes informações são trocadas entre componentes:

- I *Bitcode* representando o programa não otimizado. Essa é a saída do compilador.
- II Informa se a transformação τ está habilitada. Deverá conter nos dados gerados pelo programa informações de tempo de execução para um programa de teste onde a transformação τ está habilitada e desabilitada.
- III Vetor de características extraído de uma função do programa.
- IV *Bitcode* otimizado. Um *bitcode* é gerado para cada combinação de função do programa, transformação τ e ativação da transformação (habilitado ou desabilitado)
- V Executável otimizado.
- VI Tempo de execução de programa.

Figura 13: Geração de dados de treinamento.

VII Tempo de referência de execução do programa. O programa utilizado para a medição do tempo de referência foi aquele em que todas as transformações da Tabela 1 estavam desabilitadas para todas as funções.

VIII Dados de treinamento.

O item I da Figura 13 representa o *bitcode* do programa teste compilado sem nenhuma transformação habilitada. O Seletor de Função tinha como comportamento escolher funções de um programa para otimizá-las usando as transformações da Tabela 1 gerando os *bitcode* marcados como item IV. O item II determinará se as funções escolhidas no Seletor de Função teria a transformação τ habilitada ou desabilitada na execução atual. Cada *bitcode* é compilado gerando o item V e executado pelo sistema operacional para a medição do tempo de execução VI. Cada tempo de execução é comparado com um valor de referência para determinar se há uma modificação significativa. Todos os dados significativos foram reunidos para a produção dos dados de treinamento.

Para acelerar a execução da produção dos dados de treinamento, o experimento fez uso da biblioteca MPI (Message Passing Interface) [19]. Usando essa biblioteca foi possível paralelizar o trecho IV - V - VI da Figura 13. Cada algoritmo de treinamento também foi executado em paralelo também utilizando MPI.

6.3 Seleção de Exemplos

O Seletor de Exemplos Significativos da Figura 13 precisava ser capaz de determinar se uma modificação no plano de compilação (ativação ou desativação de uma transformação para uma função) produzia alguma alteração significativa no tempo de execução do programa.

A medida do tempo de execução do programa estava sujeita a diversas interferências exercidas pelo sistema operacional. Devido ao sistema operacional permitir a execução de vários programas simultaneamente, possíveis competições por recursos entre programas produzem oscilações nas medições de tempo. Mesmo se uma máquina fosse alocada unicamente para a execução de um programa, haveria competição entre o programa em otimização e programas que integram o próprio sistema operacional.

O procedimento estatístico escolhido para julgar se dois tempos de execução apresentavam valores diferentes foi baseado na representação visual usando *boxplots* (e por consequência medianas e quartis). A vantagem de utilizar *boxplots* é que se baseiam em medianas (valor pouco afetado por valores discrepantes [8]) e por possuírem uma técnica de remoção de valores discrepantes: sendo Q1 e Q3 o primeiro e o terceiro quartis, m a mediana, então um valor t de tempo é considerado discrepante se $|t - m| > 1.5 * (Q3 - Q1)$.

A Figura 14 mostra os possíveis *boxplots* comparando um valor de referência com um valor de teste. Os valores a , b e x , y são os 1º e 3º quartis dos valores de referência e de teste respectivamente.

Sabendo que os valores a e b da Figura 14 representam o 1º e 3º quartis da medida de referência e que os valores x e y representam o 1º e 3º quartis da medida de teste, então, dada uma taxa de melhoria δ , as seguintes equações podem ser aplicadas a cada caso:

Figura 14: Seletor de exemplos significativos. Os valores a e b representam o 1º e 3º quartis da medida de referência respectivamente. Os valores x e y representam o 1º e 3º quartis da medida de teste. Apenas nos gráficos I e III podemos dizer que há uma alteração no tempo de execução do teste em relação a referência.

I Redução no tempo de execução do teste em relação à referência. Nesse caso deve-se associar a função ao rótulo **1** (ativação de transformação) nos dados de treinamento para a transformação em questão.

$$\begin{cases} x < (1 - \delta)a \\ y < (1 - \delta)b \end{cases} \Rightarrow \begin{cases} a - x > \delta a \\ b - y > \delta b \end{cases}$$

II Dados inter-quartis da referência estão contidos nos dados de teste. Não é possível afirmar se houve uma aumento ou redução no tempo de execução.

$$\begin{cases} x < a \\ y > b \end{cases}$$

III Aumento no tempo de execução do teste em relação a referência. Nesse caso deve-se associar a função ao rótulo **0** (desativação de transformação) nos dados de treinamento para a transformação em questão.

$$\begin{cases} x > (1 + \delta)a \\ y > (1 + \delta)b \end{cases} \Rightarrow \begin{cases} a - x < -\delta a \\ b - y < -\delta b \end{cases}$$

IV Dados inter-quartis de teste estão contidos nos dados de referência. Não é possível afirmar se houve uma aumento ou redução no tempo de execução.

$$\begin{cases} a < x \\ b > y \end{cases}$$

6.4 Aprendizado de Máquina

Devido a representação intermediária de funções serem representadas por estruturas não-numéricas (grafos), foram extraídas características numéricas aproximadas dessa representação para produzir o vetor de características. Essa extração será feita através da contagem de instruções ou blocos de instruções presentes nessa estrutura. Foi utilizada a própria divisão de tipos de instruções presentes

instrs	Número de instruções
binInstrs	Número de instruções de dois operandos
memInstrs	Número de instruções de acesso a memória
termInstrs	Número de instruções terminal
castInstrs	Número de instruções de conversão
otherInstrs	Número de instruções que não são binárias, de memória, terminais ou de conversão
basicBlks	Número de blocos básicos

Tabela 2: Vetor de Características Utilizado. Esse vetor de características é baseado na classificação de tipos de instruções presente na LLVM.

na LLVM para produzir o vetor de características, apresentado na Tabela 2. Esse método é o mesmo utilizado em [9, 25], e, assim como nesses trabalhos, não é assegurada nenhuma garantia de que esse é o melhor vetor de características.

A partir dos exemplos gerados no processo de busca, aplicou-se algoritmos de aprendizado para gerar o processo de elaboração do plano de compilação. Os algoritmos de aprendizado utilizados foram a regressão logística, SVM e árvore de decisão que foram identificados nos resultados por **LOGIT**, **SVM** e **TREE** respectivamente.

A biblioteca *scikit-learn* [24] foi usada no aprendizado. Cada algoritmo de aprendizado possui parâmetros particulares que podem ser configurados. Nesse trabalho usou-se os valores padrões sugeridos pelo *scikit-learn* para cada um dos algoritmos (Tabela 3).

6.5 Programas de Treinamento e Teste

O *benchmark* SPEC cpu2000 [14] foi utilizado tanto para treinamento quanto para testes (um programa ou era de treinamento ou era de teste). Dentro do cpu2000 há dois tipos de *benchmark*: O CINT2000, para programas que realizam computações sobre números inteiros e o CFP2000 que realiza computações sobre números de ponto flutuante. Os programas do CINT2000 podem ser vistos na

Algoritmo	Parâmetros
LOGIT	<pre>LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True, intercept_scaling=1, penalty='l2', random_state=None, tol=0.0001)</pre>
SVM	<pre>SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)</pre>
TREE	<pre>DecisionTreeClassifier(compute_importances=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_density=None, min_samples_leaf=1, min_samples_split=2, random_state=None, splitter='best')</pre>

Tabela 3: Parâmetros dos Algoritmos de Aprendizado. Parâmetros padrões usados pela biblioteca *scikit-learn* no formato da linguagem de programação *Python*.

Programa	Uso	Descrição
gzip	Treino	Compressão
vpr	Teste	Roteamento e posicionamento para FPGAs
gcc	Teste	Compilador C
mcf	Treino	Otimização combinatória
crafty	Treino	Jogador de Xadrez
parser	Treino	Processamento de palavras
eon	Treino	Computação gráfica
perlbml	Treino	Interpretador da linguagem de programação <i>Perl</i>
gap	Teste	Teoria dos grupos e interpretador
vortex	Treino	Banco de dados orientado a objetos
bzip2	Teste	Compressão
twolf	Teste	Roteamento e posicionamento em circuitos integrados

Tabela 4: Programas do CINT2000.

Tabela 4 e os do CFP2000 podem ser vistos na Tabela 5.

A comparação dos algoritmos de aprendizado será feita através do desempenho da elaboração de planos de compilação sobre programas de teste. Na Tabela 4 e na Tabela 5 a coluna central informa se o programa foi usado como treinamento ou teste. Os programas escolhidos como treinamento e teste não seguiram nenhum critério especial, foram praticamente escolhidos de maneira aleatória. Por razões práticas, alguns testes cujo tempo de execução era muito alto (demorado) foram colocados como teste.

Cada programa do *benchmark* SPEC cpu2000 possui três diferentes tipos de dados de entrada: *test*, *train* e *ref*. As entradas do tipo *test* apresentam o menor tempo de execução enquanto que as entradas do tipo *ref* apresentam o maior tempo de execução. A execução do experimento adotou as entradas do tipo *test* para evitar uma grande demora na execução. Caso fosse necessário bastaria alterar um arquivo de configuração para trocar o tipo de entradas que os programas utilizariam.

Programa	Uso	Descrição
wupwise	Teste	Cromodinâmica quântica
swim	Treino	Modelagem de águas rasas
mgrid	Teste	Método <i>multigrid</i> para campos potenciais 3D
applu	Treino	Mecânica dos fluidos computacional
mesa	Teste	Gráficos 3D
galgel	Treino	Mecânica dos fluidos computacional
art	Treino	Rede neural e reconhecimento de imagens
equake	Teste	Simulação de propagação de ondas sísmicas
facerec	Teste	Sistema de reconhecimento de face
ammp	Treino	Química computacional
lucas	Treino	Teoria dos números
fma3d	Teste	Simulação de colisões usando elementos finitos
sixtrack	Teste	Projeto de acelerador de partículas
apsi	Treino	Meteorologia: distribuição de poluentes

Tabela 5: Programas do CFP2000.

7 RESULTADOS

Nessa seção são apresentados os resultados da aplicação do plano de compilação sobre programas de teste. Na primeira seção é feita uma análise do efeito de múltiplas execuções do experimento enquanto que, nas seções restantes, são mostrados os resultados do experimento para os programas de teste (na Tabela 4 e na Tabela 5 estão indicados os programas que foram utilizados como teste).

Os resultados são apresentados como *boxplots* do tempo de execução para se comparar a otimização padrão da LLVM, o código sem otimização e otimizações envolvendo diferentes algoritmos de aprendizado (os algoritmos de regressão logística, SVM e árvores de decisão são identificados por **LOGIT**, **SVM** e **TREE** respectivamente).

A partir desses gráficos podemos responder se utilizando aprendizado de máquina podemos obter um melhor desempenho do programa gerado em relação a LLVM e qual algoritmo de aprendizado é melhor. Sabendo que a comparação é feita entre tempos de execução, o conceito de **melhor plano de compilação** nesse caso é aquele que apresenta o **menor tempo de execução**. O tempo usado é o tempo total de execução do programa.

Os dados foram obtidos em dois ambientes de execução distintos. O primeiro local era um *cluster* e o segundo um computador de mesa (PC). Para cada teste são mostrados dados coletados em cada ambiente de execução no decorrer desse capítulo. A especificação dos ambientes de execução pode ser vista na Tabela 6.

Ambiente	Especificação
<i>Cluster</i>	20 CPUs com 512 GB e processador Xeon(R) CPU E7-2870 @ 2.40GHz. Scientific Linux 6.x.
PC	processador Pentium D CPU 3.40GHz, 3.2 GB de memória e Ubuntu Linux 14.10

Tabela 6: Especificação dos Ambientes de Execução.

Apenas o programa "eon" não foi considerado no processo de treinamento. Não foi possível executá-lo no ambiente de *cluster* embora fosse possível executá-lo no PC. Para uma comparação justa, o programa "eon" foi removido do experimento em ambos os ambientes de execução.

7.1 CINT2000

Nessa seção são apresentados o tempo de execução para os programas de teste "gap"(Figura 15), "twolf"(Figura 16), "vpr"(Figura 17), "gcc"(Figura 18) e "bzip2"(Figura 19) para os ambientes de execução *cluster* e PC.

7.2 CFP2000

Nessa seção são apresentados o tempo de execução para os programas de teste "wupwise"(Figura 20), "mesa"(Figura 21), "equake"(Figura 22), "fma3d"(Figura 23), "facerec"(Figura 24), "sixtrack"(Figura 25) e "mgrid"(Figura 26) para os ambientes de execução *cluster* e PC.

7.3 Efeitos de Múltiplas Execuções

Para verificar a confiabilidade da medida do tempo de execução foram construídos os gráficos das Figuras 27 e 28 para o *cluster* e PC respectivamente a partir da execução do experimento três vezes. Nessa figuras são mostrados o tempo de execução mediano para 10 execuções dos programas de teste compila-

Figura 15: Tempo de execução para o programa gap.

Figura 16: Tempo de execução para o programa twolf.

Figura 17: Tempo de execução para o programa vpr.

Figura 18: Tempo de execução para o programa gcc.

Figura 19: Tempo de execução para o programa bzip2.

Figura 20: Tempo de execução para o programa wupwise.

Figura 21: Tempo de execução para o programa mesa.

Figura 22: Tempo de execução para o programa equake.

Figura 23: Tempo de execução para o programa `fma3d`.

Figura 24: Tempo de execução para o programa `facerec`.

Figura 25: Tempo de execução para o programa sixtrack.

Figura 26: Tempo de execução para o programa mgrid.

Figura 27: Tempo de execução mediano para três execuções do experimento no *cluster*.

dos com o plano de compilação padrão da LLVM e com planos de compilação usando o aprendizado de máquina.

Para as medidas coletadas no *cluster* é possível notar que no programa six-track há uma grande diferença em uma das medidas enquanto que nas medidas coletadas no PC pouca variação nos valores medianos é observável.

Por causa da discrepância no valor mediano do tempo de execução no *cluster* foi feito o seguinte experimento extra: executou-se o programa sixtrack em 5 momentos distintos usando o plano de compilação padrão da LLVM por 100 vezes no *cluster* e no PC (Figura 29). Para o *cluster* é possível verificar que mesmo com um aumento no número de execuções não foi possível obter uma tendência nos dados.

Figura 28: Tempo de execução mediano para três execuções do experimento no PC

Figura 29: Mediana do tempo de execução do programa sixtrack executado 100 vezes em 5 momentos distintos.

8 ANÁLISE DOS RESULTADOS

O gráfico da Figura 29 mostra que no *cluster*, mesmo executando o experimento 100 vezes, não é possível obter confiabilidade na medição de tempo (na Figura 27 ocorreu uma medida de tempo muito diferente de outros experimentos). Tal falta de confiabilidade deve-se, provavelmente, ao fato de que no PC foi possível garantir exclusividade de uso enquanto que no *cluster* o experimento estava em um ambiente compartilhado com outros processos. Esta seção analisará apenas aos dados obtidos no ambiente de execução do PC.

Para determinar qual otimização foi melhor que a otimização padrão da LLVM construiu-se as Tabelas 7 e 8 para os programas dos grupos CINT (Tabela 4) e CFP (5) respectivamente. Essas tabelas foram construídas usando as figuras de 15 até 26. A melhoria, valor e rótulo presentes nas células das tabelas 7 e 8, foi calculada utilizando a seguinte fórmula:

$$melhoria(prg, alg) = \frac{m(prg, alg) - m_{LLVM}(prg)}{m_{LLVM}(prg)}$$

onde os parâmetros *alg* representa o algoritmo ($alg \in \{\text{LOGIT}, \text{SVM}, \text{TREE}\}$), *prg* representa o programa de teste (programas das tabela 4 e 5), $m_{LLVM}(prg)$ a mediana dos tempos de execução coletados usando o plano de compilação padrão da LLVM para um programa *prg* e $m(prg, alg)$ a mediana dos tempos de execução para o programa *prg* usando o algoritmo de aprendizado *alg*.

Programa	LOGIT	SVM	TREE
gap	Menor (−2.52%)	Menor (−2.12%)	Menor (−1.59%)
twolf	Maior (+2.40%)	Maior (+2.43%)	Maior (+2.34%)
vpr	Maior (+4.51%)	Maior (+1.71%)	Maior (+3.27%)
gcc	Maior (+1.47%)	Maior (+2.07%)	Maior (+1.65%)
bzip2	Igual (0.00%)	Igual (0.00%)	Igual (0.00%)

Tabela 7: Resultado das otimizações sobre os programas CINT. Os valores "maior", "menor" e "igual" para os três tipos de otimização indicam, respectivamente, que o tempo de execução foi maior, menor ou igual ao programa de teste quando compilado com o otimizador padrão da LLVM.

Programa	LOGIT	SVM	TREE
wupwise	Igual (0.00%)	Igual (0.00%)	Igual (0.00%)
mesa	Igual (0.00%)	Maior (+4.58%)	Menor (−1.89%)
equake	Maior (+6.59%)	Maior (+6.51%)	Maior (+6.74%)
fma3d	Menor (−1.27%)	Igual (0.00%)	Igual (0.00%)
facerec	Igual (0.00%)	Igual (0.00%)	Igual (0.00%)
sixtrack	Igual (0.00%)	Maior (+1.10%)	Igual (0.00%)
mgrid	Igual (0.00%)	Igual (0.00%)	Igual (0.00%)

Tabela 8: Resultado das otimizações sobre os programas CFP. Os valores "maior", "menor" e "igual" para os três tipos de otimização indicam, respectivamente, que o tempo de execução foi maior, menor ou igual ao programa de teste quando compilado com o otimizador padrão da LLVM.

Se a $|melhoria(prg, alg)| < 0.01$ os tempos de execução foram iguais, o programa em ambas as configurações apresentou o mesmo tempo de execução. Se $melhoria(prg, alg) < -0.01$ então o tempo de execução do programa usando a otimização com aprendizado de máquina foi menor. Se $melhoria(prg, alg) > 0.01$ o tempo de execução do otimizador proposto foi maior que o padrão da LLVM.

Para testes do CINT (Tabela 7) é possível verificar que apenas para o teste "gap" foi possível obter uma diminuição no tempo de execução em torno de 1.5%. O teste "bzip2" não apresentou mudança significativa. Olhando para os demais testes ("twolf", "vpr" e "gcc"), é possível notar que houve um aumento no tempo de execução na ordem de 2%. Em todos os testes o tempo de execução ficou próximo (menos de 7% de aumento) ao tempo de execução do programa compilado com o plano padrão da LLVM.

Para testes do CFP (Tabela 8) é possível verificar que os testes "wupwise", "facerec" e "mgrid" não apresentaram diferenças significativas com relação ao plano de compilação padrão da LLVM. O teste "mesa" apresentou um aumento no tempo de execução quando usou SVM e uma diminuição quando usou árvores de decisão. O teste "fma3d" foi melhor quando gerou o plano de compilação usando regressão logística e apresentou o mesmo resultado para os demais algoritmos de aprendizado. O teste "equake" apresentou um aumento no tempo de execução em torno de 6.5%. No "sixtrack" a SVM produziu um modelo com maior tempo de execução enquanto que os demais algoritmos de aprendizado ficaram próximos ao plano padrão da LLVM. Assim como nos testes do CINT, os tempos de execução quando usado aprendizado de máquina ficaram próximos dos compilados com o plano de compilação da LLVM.

Como os dados das Tabelas 7 e 8 sugerem que o otimizador usando aprendizado de máquina se aproxima do plano de compilação padrão da LLVM, é preciso verificar se os dados de treinamento apenas contém exemplos positivos. Caso só haja exemplos positivos, estará tentando aprender uma função cujo resultado é sempre 1, ativando a otimização para todos os casos, que é exatamente o que o plano padrão de compilação da LLVM usa.

Na Tabela 9 é possível verificar que cada transformação contém exemplos positivos e negativos nos dados de treinamento. Para todas as transformações é possível garantir que, em pelo menos 30% dos dados de treinamento, desabilitar a transformação sobre uma função resultará em uma diminuição no tempo de execução.

Uma simples análise do vetor de características pode ser feita aplicando o PCA sobre os dados de treinamento (Tabela 10). Os componentes *instrs*, *binInstrs*, *memInstrs*, *termInstrs*, *castInstrs*, *otherInstrs* e *basicBlks* do vetor de características são transformados nos componentes w_1 , w_2 , ..., w_7 respectivamente.

Otimização	Positivos (%)	Negativos (%)
Aggressive Dead Code Elimination	63.24	36.76
Combine redundant instructions	63.68	36.32
MemCpy otimização	65.15	34.85
Dead Store Elimination	64.18	35.82
Simplify the CFG	63.32	36.68
Value Propagation	64.18	35.82

Tabela 9: Análise dos dados de treinamento. As colunas "Positivos" e "Negativos" representam a porcentagem de exemplos onde a transformação habilitada e desabilitada resultam em um menor tempo de execução.

transformação	Porcentual de variância por componente (%)						
	w_1	w_2	w_3	w_4	w_5	w_6	w_7
Aggressive Dead Code Elimination	96.17	3.31	0.31	0.14	0.07	0.00	0.00
Combine redundant instructions	96.02	3.49	0.31	0.12	0.06	0.00	0.00
MemCpy otimização	96.42	3.11	0.30	0.11	0.06	0.00	0.00
Dead Store Elimination	96.16	3.32	0.31	0.14	0.07	0.00	0.00
Simplify the CFG	96.15	3.31	0.35	0.13	0.06	0.00	0.00
Value Propagation	96.16	3.32	0.31	0.14	0.07	0.00	0.00

Tabela 10: Análise do Vetor de Características. Cada valor representa a porcentagem da variância de um componente em relação ao total.

Usando a transformação proposta pelo PCA poder-se-ia reduzir o vetor de características para todas as transformações a dois componentes: w_1 e w_2 . Isso melhoraria o desempenho dos algoritmos de aprendizado já que diminuiria o número de dimensões (o aumento do número de dimensões prejudica a qualidade dos resultados obtidos [20, 7]).

9 CONCLUSÃO

Os planos de compilação obtidos usando o aprendizado de máquina com o vetor de características proposto se aproximou do padrão utilizado pela LLVM embora não foi possível ultrapassá-lo (apenas em um *benchmark* foi possível obter uma redução no tempo de execução).

Em [25] não foi possível obter uma melhoria em relação ao plano de compilação ajustado manualmente durante anos pela máquina virtual utilizada mas foi possível obter uma diminuição no tempo de inicialização da máquina virtual. Nesse trabalho a linguagem Java permitiu extrair um vetor de características maior por possuir um número maior de construções que a representação intermediária da LLVM e o processo de busca de planos de compilação partiu de alterações aleatórias sobre um plano de compilação existente.

Planos de compilação individuais por método deram um aumento de 1% a 2% em relação ao plano O1 em [9]. No plano de compilação O2 não foi observada nenhuma melhoria. Os resultados desse trabalho não mostraram uma vantagem significativa para planos individuais por função embora o uso do aprendizado de máquina aproximou-se do plano de compilação padrão da LLVM. Mesmo não produzindo vantagem na execução do programa, esse trabalho permite que o processo de produção de heurísticas seja automatizado.

O trabalho de [22] conseguiu garantir um ganho em relação a transformação padrão de *loop unrolling* usando aprendizado de máquina. Trabalhos seguindo

a linha de [17] onde é feita a geração do vetor de características usado algoritmos genéticos podem ser uma alternativa para a busca de um melhor vetor de características sobre a representação intermediária da LLVM.

Os programas de teste e de treino foram escolhidos de forma arbitrária. Não se pode garantir que qualquer outra divisão de programas devolveriam o mesmo resultado.

Pouco se tirou de vantagem da possibilidade de executar o levantamento de dados sob um *cluster* (usando MPI) devido ao ruído presente na medição de tempo. Esse ruído foi devido, provavelmente, à impossibilidade de garantir o uso exclusivo do *cluster*.

Uma alternativa para a medição de tempo total do programa era medir o tempo de execução sobre cada uma das funções do programa. O método de geração de dados usado nesse trabalho não é capaz de detectar pequenas melhorias sobre funções pouco executadas já que essas funções contribuirão pouco para o tempo de execução total do programa. Possíveis melhores planos de compilação sobre funções específicas podem passar despercebidos devido ao ruído na medição de tempo.

Inicialmente tentou-se usar um esquema de compilação usando o compilador JIT presente na LLVM mas devido ao problema que, quando um programa sob testes finalizava, encerrava também o programa que estava coletando os dados optou-se por compilação tradicional. Na compilação tradicional o programa sobre teste era executado em um processo separado e coletado seu tempo de execução para ser utilizado no processo de aprendizado.

O processo utilizado para a aplicação de aprendizado de máquina em compiladores sobre a LLVM é o legado desse trabalho, já que trabalhos anteriores focavam outros arcabouços (geralmente máquinas virtuais). Resultados melhores

podem ser obtidos apenas realizando extensões (incluindo direcionar para áreas inexploradas).

O uso da LLVM tem como vantagem a possibilidade de aplicar o processo de otimização usando aprendizado de máquina a qualquer programa que possa ser convertido em *bitcode* facilitando a evolução de protótipo a ferramenta.

9.1 Trabalhos Futuros

O programa que executava testes inicialmente foi projetado de forma a permitir que os planos de compilação O1, O2 e O3 (do menor número de transformações para o maior) fossem utilizados, mas usou-se apenas o plano O1 e as transformações da Tabela 1 foram utilizadas. Um trabalho futuro seria verificar o efeito do aprendizado de máquina para um número maior de transformações.

O vetor de características proposto por esse trabalho (Tabela 2) foi o mais simples possível para se iniciar qualquer tipo de experimento. Os dados da Tabela 10 são um bom indicativo que esse vetor pode ser melhorado. Um bom trabalho futuro seria pesquisar um método para gerar automaticamente esse vetor de características, partindo talvez de [17].

Assim como o vetor de características, a ativação e desativação de transformações sobre funções era a parte do plano de compilação mais simples de se modificar. Lembrando que a LLVM possui outros tipos de passo (sobre laços e grafo de chamada por exemplo), o aprendizado de máquina pode ser aplicado sobre outros tipos de passo, não necessariamente para a ativação e desativação mas para a determinação de valores numéricos (usado em heurísticas) como limiar para realizar o *inline* de funções e número de replicações de laços no *loop unrolling*.

Uma possível utilidade para a ativação e desativação de função no plano

de compilação é para a redução do tempo de compilação. O tempo total de compilação não foi medido nesse experimento, mas apresenta pontencial para redução. Em 30% dos casos não aplicar a transformação resultaria em uma melhoria no tempo de execução do programa de teste (Tabela 9). Redução de tempo que seria interessante para um compilador JIT.

Determinar o que seria um bom programa de treino e de teste é outra possibilidade de trabalho futuro. A divisão utilizada nesse trabalho foi praticamente arbitrária podendo ser simplesmente sorte o resultado do experimento usando aprendizado de máquina ter se aproximado da compilação padrão da LLVM. Uma melhoria simples seria variar os programas de treino e teste usando o método *k-fold* [20, 11, 23, 7].

Poder-se-ia refazer o mesmo experimento utilizando programas de teste e treino específicos de determinada área (visão computacional, computação quântica, etc.). Esse experimento seria capaz de esclarecer se o otimizador usando aprendizado de máquina não apresentou melhoria em relação ao padrão devido ao uso de programas sem enfoque de área como o SPEC.

Partindo de um melhor vetor de características, seria interessante determinar qual algoritmo de aprendizado apresenta o melhor desempenho para a elaboração do plano de compilação. Os dados observados nesse trabalho não são suficientes para assegurar que a regressão logística (o modelo mais simples quando comparado com SVM e árvore de decisão) é o melhor classificador para a elaboração de planos de compilação.

REFERÊNCIAS

- [1] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. See <http://www.iwi.hs-karlsruhe.de>.
- [2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Alternative eText Formats Series. ADDISON WESLEY Publishing Company Incorporated, 2007.
- [3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.
- [4] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.
- [5] P. Berube and J. N. Amaral. Combined profiling: A methodology to capture varied program behavior across multiple inputs. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 210–220. IEEE, 2012.
- [6] P. Berube, A. Preuss, and J. N. Amaral. Combined profiling: practical collection of feedback information for code optimization. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, pages 493–498. ACM, 2011.
- [7] C. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.
- [8] W. Bussab and P. Morettin. *Estatística básica*. Saraiva, 2013.
- [9] J. Cavazos and M. F. P. O’Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’06*, pages 229–240, New York, NY, USA, 2006. ACM.
- [10] K. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2011.
- [11] R. Duda, P. Hart, and D. Stork. *Pattern classification*. Pattern Classification and Scene Analysis: Pattern Classification. Wiley, 2001.
- [12] M. Hall, D. Padua, and K. Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, Feb. 2009.

- [13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2001.
- [14] J. L. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, August 2000.
- [15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
- [16] C. Lattner and V. Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.
- [17] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 81–91, Washington, DC, USA, 2009. IEEE Computer Society.
- [18] G. Li, I. Ghosh, and S. P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *Computer Aided Verification*, pages 609–615. Springer, 2011.
- [19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, version 1.3 edition, 2008.
- [20] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
- [21] T. Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, 2011.
- [22] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMS '02, pages 41–50, London, UK, UK, 2002. Springer-Verlag.
- [23] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.
- [24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

- [25] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 257–266, Washington, DC, USA, 2011. IEEE Computer Society.
- [26] Y. Srikant and P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2007.
- [27] J. Stanier and D. Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, July 2013.
- [28] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, May 2003.
- [29] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. In *ACM Sigplan Notices*, volume 46, pages 295–305. ACM, 2011.
- [30] N. Wirth. *Compiler construction*, volume 1. Addison-Wesley Reading, 1996.
- [31] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *ACM SIGPLAN Notices*, 47(1):427–440, 2012.