Predição de Tempo de Compilação e Tamanho de Código em Máquinas Virtuais

Jorge Augusto Sabaliauskas
Escola Politécnica da Universidade de São Paulo
Email: jaugustosaba@usp.br

Ricardo Luis de Azevedo da Rocha
Escola Politécnica da Universidade de São Paulo
Email: rlarocha@usp.br

***Resumo*—O acelerado desenvolvimento de novos processadores faz com que os compiladores tenham que se adaptar continuamente. Para se adaptarem mais rápido começou-se a investigar a aplicação de técnicas de aprendizagem computacional nos compiladores. Este artigo visa introduzir a aplicação de técnicas de aprendizagem em compilação ao fazer a predição do tempo de compilação e o tamanho de código gerado para métodos na máquina virtual Maxine utilizando características extraídas do *bytecode*.**

I. Introdução

O processo manual de ajuste do compilador faz com que os benefícios de uma nova geração de processadores não sejam aproveitados imediatamente. Um processo de adaptação automático as novas arquiteturas é desejável.

O aprendizado computacional procura tentar fazer o computador aprender a executar tarefas complexas sem que um algoritmo precise ser explicitamente determinado.. Há dois modos principais de aprendizado: supervisionado e não-supervisionado. Algoritmos de aprendizado supervisionado tentam aprender um modelo através de exemplos enquanto que os algoritmos de aprendizado não-supervisionado tentam descobrir padrões nos dados geralmente por identificação de semelhanças. Combinações dos aprendizados supervisionados e não-supervisionados também podem ser empregadas.

No início dos anos 2000 ocorreram com maior frequência aplicações das técnicas de aprendizagem computacional na área de compiladores para a criação de um processo adaptativo de seleção de otimizações (por exemplo o artigo [1]). Heurísticas utilizadas anteriormente eram produzidas através de um especialista que manualmente trabalhava sobre o código gerado pelo compilador em busca de melhores oportunidades de geração de código [2].

O objetivo deste trabalho é mostrar como técnicas de aprendizagem computacional podem ser aplicadas no contexto de compiladores através de duas predições simples: tempo de compilação e tamanho de código. Para a demonstração foi escolhida a máquina virtual Java Maxine [3] e o *benchmark* DaCapo [4].

Essa predição é feita sobre o tempo de compilação e tamanho de código gerado sobre métodos de uma classe Java. Características do método são calculadas estaticamente (em tempo de compilação) e então usadas na predição. Essa predição é feita levando em conta os diferentes perfis de compilação disponíveis na Maxine.

Como algoritmo de aprendizado é utilizado o algoritmo de regressão linear, um algoritmo de aprendizagem supervisionado. Esse tipo de regressão foi escolhido por ser o mais simples e ideal para uma introdução. Nesse algoritmo a saída consistirá de uma combinação linear das características do método. Para validação é utilizado o algoritmo *k-fold validation*.

O restante desse artigo organiza-se da seguinte forma:

- Em Metodologia é apresentado como será realizado o aprendizado
- Em Experimento é descrita a preparação do experimento
- Em Resultados é mostrado os preditores obtidos
- Em Discussão os resultados são comparados com o modelo intuitivo
- Em Conclusão será abordada a importância dos resultados obtidos

II. Metodologia

Em compiladores é comum possuir quatro níveis de compilação identificados por O0, O1, O2 e O3. O perfil de compilação indicado por O0 não aplica nenhuma otimização enquanto que o perfil O3 aplica todas as otimizações disponíveis. Cada otimização requer um tempo de compilação e gera um código de determinado tamanho.

Tanto o tempo de compilação quando o tamanho do código gerado pela compilação podem ser determinados em função do trecho do programa alvo do processo de compilação.

Como tanto o tempo de compilação quanto o tamanho de código gerado são valores numéricos, o algoritmo de aprendizado utilizado é chamado de regressão. O algoritmo de regressão utilizado é o de regressão linear (maiores detalhes em [5]) descrito por (1), onde h_θ (chamado comumente de hipótese) pode ser tanto o tempo de compilação quanto o tamanho do código gerado, n é o número de características, x_i tal que $i \in \{1 \dots n\}$ são as características computadas sobre os métodos, x_0 é o valor constante 1 e θ_i tal que $i \in \{0 \dots n\}$ é o vetor de pesos.

$$h_\theta(x) = \theta^T X = \sum_{i=0}^{n} \theta_i x_i \quad (1)$$

Tabela I. CARACTERÍSTICAS CALCULADAS SOBRE OS BYTECODES DE MÉTODOS.

x_i	**Característica**	**Descrição**
1	bytecodes	Número de bytecode em um método
2	localSpace	Espaço para locais
3	synch	O método é sincronizado ?
4	exceptions	O método possui código de tratamento de instruções ?
5	leaf	O método é folha (sem chamadas para outras instruções) ?
6	final	O método é declarado final ?
7	private	O método é declarado privado ?
8	aload_astore	Fração instruções aload e astore no bytecode
9	primitive_long	Fração de instruções sobre primitivos ou longs
10	compare	Fração de bytecodes de comparação
11	jsr	Fração de instruções JSR
12	switch	Fração de instruções switch
13	putOrGet	Fração de instruções put ou get
14	invoke	Fração de instruções de invocação
15	new	fração de instruções de criação de instâncias
16	arrayLength	fração de instruções ArrayLength
17	athrow_checkCast_monitor	fração de instruções athrow, checkcast or monitor
18	multi_newarray	fração de bytecodes multiNewArray
19	simple_long_real	fração de bytecodes de conversão sobre tipos simples, long ou real

Para definir qual é a melhor reta na regressão linear é necessário definir a função de custo. A função de custo utilizada nesse artigo é a *mean squared error* (de uso mais comum) descrita por (2) onde $X^{(i)}$ e $Y^{(i)}$ representam o vetor de características $(1, x_1, \ldots, x_n)$ e o valor observado respectivamente na amostra rotulada com índice i e m é o total de amostras.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(X^{(i)}) - Y^{(i)})^2 \tag{2}$$

A partir da equação (2) pode-se encontrar a melhor reta através do algoritmo de descida de gradiente. Logo, o vetor de pesos pode ser encontrado por (3), onde α é chamado de coeficiente de aprendizado. A escolha de α deve ser feito com cautela, pois se for um valor muito alto, o algoritmo pode não convergir. Pode-se detectar a não convergência monitorando caso função de custo (2) não diminua de uma iteração para outra.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j \tag{3}$$

A partir do trecho do programa podem ser calculadas características que descrevem o trecho a ser compilado (nesse caso o método de uma classe). As características são obtidas a partir de uma simples passagem pelo *bytecode* coletando informações sobre instruções, como o feito em [6]. Todas as características computadas sobre métodos são descritas na Tabela I. Alguns exemplos do resultado do cálculo de características podem ser vistos na Tabela VII.

Para o processo de treinamento será utilizado o método *k-fold validation* onde as amostras de treinamento são divididas em k partições, onde k-1 partições são usadas no treinamento e 1 partição para teste. Esse processo é repetido k vezes para poder estimar o erro do algoritmo de aprendizado (nesse caso a regressão linear).

Foram retirados três diferentes conjuntos de treinamento, um para cada perfil de compilação (O1, O2 e O3). No final foram produzidas seis equações no formato de (1): três para predição de tempo de compilação e três para predição de tamanho de código gerado.

III. EXPERIMENTO

Para o experimento foi utilizada a máquina virtual Maxine em conjunto com o *benchmark* DaCapo, uma coleção de programas usualmente utilizados sobre a máquina virtual Java. O uso da Maxine envolvendo aprendizado computacional já foi feito em [7]. Através de configurações de linha de comando foi alterado o perfil de otimização. Para o processamento de *bytecodes* foi utilizada a biblioteca *javassist* [8].

A coleta do vetor de características foi feita através de um *Java Agent* que calculava as características conforme o carregamento de classes pela máquina virtual. Como o *benchmark* DaCapo também faz um gerenciamento do carregamento de classes, algumas classes não tiveram as características calculadas usando o *Java Agent*. Para contornar a perda de informações sobre métodos, foi criado um programa para varrer o arquivo do DaCapo em busca de classes e calcular suas características. Alguns métodos não tiveram características calculadas devidos a limitações da biblioteca *javassist* mas mesmo assim foi possível obter uma quantidade de amostras $m > 1500$ para cada perfil de compilação.

O resultado do tempo de compilação de um método e o tamanho do código gerado foram obtidos facilmente por uma configuração de linha de comando na Maxine. Ao ativar essa opção a Maxine passou a imprimir toda compilação que realizava. Essa impressão teve que ser feita em um arquivo pois senão pode-se misturar os dados de compilação com a saída do DaCapo, dificultando o processamento posterior.

Com os features para cada método calculado pelo *Java Agent* ou leitura do arquivo do DaCapo e o tempo de compilação e tamanho de código informados pela Maxine, bastou juntar esses dados para se produzir o conjunto de treinamento para cada perfil de compilação. Foram mantidas as assinaturas dos métodos por conveniência.

Para o treinamento foi utilizado a ferramenta *Weka* [9] que já possuía implementado o algoritmo de regressão linear. Na validação *k-fold validation* foi utilizado o valor k=10 sugerido pelo próprio Weka.

IV. RESULTADOS

Nessa seção podem ser encontrados o resultado do treinamento da regressão linear no *Weka* para o tamanho de código gerado e tempo de compilação. Além disso são fornecidas cinco previsões de saída para dados do conjunto de treinamento.

A. Otimização O1

O resultado para previsão do tamanho do código na otimização O1 fornecido pelo *Weka* pode ser visto na Figura

```
size =
     9.2062 * bytecodes +
    38.3063 *  localSpace +
   125.9147 *  synch +
  1245.0898 *  exceptions +
   155.9847 *  leaf +
   124.2994 *  aload_astore +
 -1484.4537 *  primitive_long +
-21011.2525 *  jsr +
   600.4225 *  invoke +
  2344.8719 *  new +
   479.8091 *  athrow_checkCast_monitor +
 -2287.1684 *  simple_long_real +
  -249.659
```

Figura 1. Regressão linear para a predição de tamanho de código no modo O1. O tamanho de código é dado em bytes.

Tabela II. CINCO PREDIÇÕES DE TAMANHO NA OTIMIZAÇÃO O1.

Valor Real (bytes)	Predição (bytes)	Erro (bytes)
1556	1843.419	287.419
1174	1693.233	519.233
32	20.907	-11.093
77	242.571	165.571
29	20.907	-8.093

IV-A. O comportamento para cinco previsões do tamanho de código pode ser vistos na Tabela II. O conjunto de treinamento possuía $m = 1843$ instâncias.

O resultado para previsão do tempo de compilação na otimização O1 fornecido pelo *Weka* pode ser visto na Figura 2. O comportamento para cinco previsões do tempo de compilação pode ser vistos na Tabela III.

B. Otimização O2

O resultado para previsão do tamanho do código na otimização O2 fornecido pelo *Weka* pode ser visto na Figura 3. O comportamento para cinco previsões do tamanho de código pode ser vistos na Tabela IV. O conjunto de treinamento possuía $m = 1657$ instâncias.

```
time =
   0.068  * bytecodes +
  -0.6754
```

Figura 2. regressão linear para predição de tempo de compilação no modo O1. O tempo é dado em milisegundos.

Tabela III. CINCO PREDIÇÕES DE TEMPO DE COMPILAÇÃO NA OTIMIZAÇÃO O1.

Valor Real (ms)	Predição (ms)	Erro (ms)
3.2	12.637	9.437
3.8	11.111	7.311
0.2	-0.515	-0.715
0.0	0.357	0.357
0.1	-0.515	-0.615

```
size =
    12.4861 * bytecodes +
    50.3098 *  localSpace +
  1500.5807 *  exceptions +
   362.1418 *  leaf +
   149.8093 *  private +
   296.6817 *  aload_astore +
 -2513.885  *  primitive_long +
 -3381.3528 *  compare +
-24723.3547 *  jsr +
  -279.414  *  putOrGet +
  1581.3714 *  invoke +
  3458.0678 *  new +
 -5519.8463 *  simple_long_real +
  -466.7764
```

Figura 3. regressão linear para a predição de tamanho de código no modo O2. O tamanho de código é dado em bytes.

Tabela IV. CINCO PREDIÇÕES DE TAMANHO NA OTIMIZAÇÃO O2.

Valor Real (bytes)	Predição (bytes)	Erro (bytes)
51	134.201	83.201
122	383.901	261.901
908	1260.888	352.888
171	575.66	404.66
383	779.047	396.047

O resultado para previsão do tempo de compilação na otimização O2 fornecido pelo *Weka* pode ser visto na Figura 4. O comportamento para cinco previsões do tempo de compilação pode ser vistos na Tabela V.

C. Otimização O3

O resultado para previsão do tamanho do código na otimização O3 fornecido pelo *Weka* pode ser visto na Figura 5. O comportamento para cinco previsões do tamanho de código pode ser vistos na Tabela VI. O conjunto de treinamento possuía $m = 1655$ instâncias.

```
time =
  0.0684 * bytecodes +
  3.5308 *  leaf +
  5.0397 *  private +
  5.2166 *  aload_astore +
 15.5741 *  invoke +
 -5.4619
```

Figura 4. regressão linear para predição de tempo de compilação no modo O2. O tempo é dado em milisegundos.

Tabela V. CINCO PREDIÇÕES DE TEMPO DE COMPILAÇÃO NA OTIMIZAÇÃO O2.

Valor Real (ms)	Predição (ms)	Erro (ms)
0.2	1.11	0.91
0.2	1.356	1.156
1.8	5.093	3.293
0.3	3.765	3.465
1	0.905	-0.095

```
size =
    12.7794 * bytecodes +
    64.3957 *  localSpace +
  2514.5824 *  exceptions +
   409.5668 *  leaf +
   168.9502 *  private +
   231.8959 *  aload_astore +
 -2680.7018 *  primitive_long +
 -3530.3777 *  compare +
-36919.3804 *  jsr +
  1915.7344 *  invoke +
  3883.0666 *  new +
 -5978.9789 *  simple_long_real +
  -587.0996
```

Figura 5. regressão linear para a predição de tamanho de código no modo O3. O tamanho de código é dado em *bytes*.

Tabela VI. CINCO PREDIÇÕES DE TAMANHO NA OTIMIZAÇÃO O3.

Valor Real (bytes)	Predição (bytes)	Erro (bytes)
114	259.124	145.124
29	-14.342	-43.342
1953	1051.965	-901.035
5672	5774.292	102.292
58	75.685	17.685

O resultado para previsão do tempo de compilação na otimização O3 fornecido pelo *Weka* pode ser visto na Figura 6. O comportamento para cinco previsões do tempo de compilação pode ser vistos na Tabela VII.

```
time =
   0.0815 * bytecodes +
   9.3518 *  exceptions +
   3.8292 *  leaf +
   1.4214 *  private +
   2.7379 *  aload_astore +
 -13.3196 *  primitive_long +
-140.6578 *  jsr +
  -2.2356 *  putOrGet +
  13.378  *  invoke +
 -39.7453 *  simple_long_real +
  -4.3945
```

Figura 6. regressão linear para predição de tempo de compilação no modo O3. O tempo é dado em milisegundos.

Tabela VII. CINCO PREDIÇÕES DE TEMPO DE COMPILAÇÃO NA OTIMIZAÇÃO O3.

Valor Real (ms)	Predição (ms)	Erro (ms)
0.3	0.716	0.416
0.1	-0.106	-0.206
3.6	3.471	-0.129
17.5	24.095	6.595
0.2	-0.477	-0.677

V. DISCUSSÃO

Um efeito indesejável de se utilizar a regressão linear foi o aparecimento de predições com valores negativos tanto para o tempo de compilação quanto para o tamanho de código, ambas deveriam ter sempre valores positivos.

Sendo tanto o tamanho do código gerado quanto o tempo de compilação diretamente proporcionais ao tamanho de *bytecodes* não é nenhuma surpresa o número de *bytecodes* aparecer em todas as fórmulas de predição. A predição de tempo na compilação O1, por exemplo, pode ser feita exclusivamente com o número de *bytecodes*.

Vale notar que as predições utilizam mais características conforme o aumento do nível de otimização. O aumento do nível indica que um maior número de otimizações estão sendo realizadas, algumas delas específicas mais relacionadas a uma categoria de instruções.

Mesmo no perfil de compilação O3 (todas as otimizações habilitadas), estão ausentes algumas características na expressão de predição de tamanho de código e tempo de compilação. Uma explicação pode ser o fato de que algumas instruções aparecem raramente no *bytecode*. Sabendo que a Maxine não possui tantas otimizações quanto uma máquina virtual comercial, é razoável supor que alguns *bytecodes* sejam desprezados.

VI. CONCLUSÃO

O processo de predição do tempo de compilação e tamanho de código gerado são um bom exemplo de como começar a aplicar técnicas de aprendizado computacional na área de compiladores (mais especificamente em máquinas virtuais).

O uso da regressão linear é interessante para fazer a predição devido a natureza do processo de compilação: aproximadamente tempo e tamanho proporcionais ao tamanho do código. Como próximo passo poderia ser escolhido um algoritmo de regressão que não produza valores negativos de tempo de compilação e tamanho de código.

É interessante notar que as características são calculadas sobre o código em tempo de compilação. Não daria para ser feita a previsão (ou pelo menos com uma boa precisão) de qualquer valor dependente da execução de um programa. A execução de um programa depende de seus dados de entrada, sem fixar a entrada não seria possível obter uma predição confiável.

A previsão do tempo de compilação e tamanho de código podem ter utilidade para o processo de adaptação em máquinas virtuais (a máquina virtual é um software adaptativo). Por exemplo, através de previsões do tempo de compilação nos diferentes perfis pode-se escolher o perfil cujo tempo de compilação não seja tão grande, lembrando que em máquinas virtuais o tempo de compilação concorre com o tempo de execução do programa.

Em trabalhos futuros deverão ser adicionadas medidas de desempenho para a melhoria das predições. Grupos de otimizações, determinados pelo perfil de compilação, podem ser substituídas por ativações individuais de otimizações. Diferenças observadas no desempenho devido a certas otimizações fornecerão informações úteis no treinamento de

preditores. O tempo de execução de métodos não foi usado nesse trabalho devido a complexidade da medição do tempo, o que necessitaria também de um tratamento estatístico mais rigoroso.

AGRADECIMENTOS

O primeiro autor agradece a bolsa de mestrado de número 33002010045P3 concedida pela CAPES.

REFERÊNCIAS

[1] 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*, ser. AIMSA '02. London, UK, UK: Springer-Verlag, 2002, pp. 41–50. [Online]. Available: http://dl.acm.org/citation.cfm?id=646053.677574

[2] 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*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 257–266. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190072

[3] (2013, Nov.) Maxine virtual machine. [Online]. Available: https://wikis.oracle.com/display/MaxineVM/Home

[4] (2013, Nov.) Dacapo benchmark. [Online]. Available: http://www.dacapobench.org/

[5] (2013, Nov.) Stanford open class. [Online]. Available: http://stanford.io/17kztSC

[6] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," *SIGPLAN Not.*, vol. 41, no. 10, pp. 229–240, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1167515.1167492

[7] D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni, "Automatic construction of inlining heuristics using machine learning," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6495004

[8] (2013, Nov.) Javassist bytecode manipulation. [Online]. Available: http://www.csg.ci.i.u-tokyo.ac.jp/ chiba/javassist/

[9] (2013, Nov.) Weka 3: Data mining software in java. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/

VII. APÊNDICE

Tabela VIII. CINCO AMOSTRAS DE CLASSES JAVA ENCONTRADAS NO CONJUNTO DE TREINAMENTO. OS VALORES DE TIME E SIZE SÃO DADOS EM MILISEGUNDOS E BYTES RESPECTIVAMENTE.

method	bytecodes	localSpace	synch	exceptions	leaf	final	private	aload_astore	primitive_long	compare	jsr	switch	putOrGet	invoke	new	arrayLength	athrow_checkCast_monitor	multi_newarray	simple_long_real	time	size
Ljava/util/LinkedList; getLast()Ljava/lang/Object;	12	2	0	0	0	0	0	0.333	0.000	0.000	0.000	0.000	0.167	0.083	0.083	0.000	0.083	0.000	0.000	0.3	169
Ljava/util/LinkedList; removeLast ()Ljava/lang/Object;	13	2	0	0	0	0	0	0.385	0.000	0.000	0.000	0.000	0.077	0.154	0.077	0.000	0.077	0.000	0.000	0.4	178
Ljava/util/Calendar; getInstance ()Ljava/util/Calendar;	10	1	0	0	0	0	0	0.300	0.000	0.000	0.000	0.000	0.200	0.300	0.000	0.000	0.000	0.000	0.000	0.4	80
Ljava/util/Date; ¡init¿ (J)V	6	3	0	0	0	0	0	0.333	0.000	0.000	0.000	0.000	0.167	0.167	0.000	0.000	0.000	0.000	0.000	0.2	64
Ljava/util/concurrent/Semaphore; tryAcquire (Ljava/util/concurrent/TimeUnit;)Z	8	4	0	0	0	0	0	0.250	0.000	0.000	0.000	0.000	0.125	0.250	0.000	0.000	0.000	0.000	0.000	0.4	92