Estimação Estática de Métricas para Distribuir Aplicações Java

Filipe Matos, António J. Esteves e João L. Sobral
Departamento de Informática, Universidade do Minho, Braga, Portugal
filipe.matos@lesium.org, esteves@di.uminho.pt, jls@di.uminho.pt

Resumo

Este trabalho apresenta um conjunto de ferramentas desenvolvidas para auxiliar a distribuição das classes de uma aplicação Java pelos recursos de uma arquitectura heterogénea, baseada em microprocessadores genéricos e dispositivos de lógica reconfigurável. Para poder tomar decisões relacionadas com essa distribuição, foi preciso identificar as diferentes formas de uma classe se relacionar com outras (fuga de referências) e obter uma estimativa estática para a complexidade dos métodos de cada classe. Os resultados obtidos com o caso de estudo RayTracer mostraram que a métrica de complexidade, estimada estaticamente, segue o mesmo padrão que o tempo de execução medido dinamicamente.

1 Introdução

O presente trabalho, integrado no projecto PPC-VM [1] [1], tem por objectivo auxiliar a distribuição de aplicações Java de elevada dimensão por uma arquitectura distribuída e heterogénea. A arquitectura será composta por diversas máquinas baseadas em microprocessadores genéricos e por uma ou mais plataformas reconfiguráveis baseadas em FPGAs Virtex-4.

Como parte integrante do processo de análise de aplicações JAVA, a análise estática permitirá estimar métricas associadas à complexidade dos métodos e à relação entre classes. Tratando-se de uma análise estática sobre a linguagem de programação JAVA, o objecto de análise pode ser o código fonte ou o código compilado (bytecodes), tendo-se optado neste caso pela análise dos bytecodes. Ao optar-se por uma análise ao nível dos bytecodes, restringiu-se as alternativas de análise ao conjunto de instruções da JVM, ou seja, a aproximadamente duas centenas de possibilidades. Contudo, esta escolha implicou uma maior dificuldade na identificação de blocos código específicos.

2 Fuga de referências

Um estudo prévio sobre diversos *profilers open-source* de Java mostrou que eles não resolvem a questão da fuga de referências. Deste modo, foi desenvolvido um algoritmo para identificar as sete maneiras de uma classe se poder relacionar com outra.

A maneira mais explícita de se obter uma referência para uma instância de uma classe, é criando-a (**tipo 1**). Quando aparece uma instrução `new` no código da aplicação, esta associa-se obrigatoriamente a uma classe, para que a maquina virtual consiga obter uma referência para uma nova instância desta.

Uma classe também se relaciona com outras quando inclui explicitamente referências para essas classes nas suas variáveis de classe (**tipo 2**) ou quando acede a variáveis de classe de outra instância (**tipo 3**). O facto de uma classe aceder a variáveis de classe de uma instância da classe diferente desta, pode ser identificado, ao nível dos bytecodes, através das instruções `putfield`, `getfield`, `putstatic` e `getstatic`.

Como a análise é efectuada por método, é possível, a partir da sua assinatura, saber quais as referências que este recebe como parâmetro (**tipo 4**) ou que passa como retorno (**tipo 5**). Para tal, foi desenvolvido um parser que identifica tanto o retorno associado a um método, como os possíveis argumentos deste. Deste modo, para obter todas as dependências, é percorrida toda a lista de argumentos, identificando aqueles que se referem a classes diferentes da classe analisada.

Por fim, quando um método faz uma invocação a um método externo à sua classe, este pode passar referências como argumentos (**tipo 6**) ou receber uma referência como retorno do método invocado (**tipo 7**). No decorrer da análise dos bytecodes são interceptadas todas as instruções que efectuem qualquer tipo de invocação (`invokeinterface`, `invokespecial`, `invokevirtual`, `invokestatic`).

3 Métrica de complexidade

A questão da dependência entre classes foi solucionada desenvolvendo uma aplicação baseada em *profilers* analisados [2] [3] [4]. Para a análise ficar completa, é preciso determinar a complexidade dos métodos. Como a maioria dos métodos segue uma linha de execução com saltos, desenvolveu-se uma aplicação que identifica (i) os saltos no código e (ii) se estes dependem directa ou indirectamente dos argumentos do método, ao qual o código se refere [5].

Para identificar os blocos de código referentes a condições ou a saltos, não foi necessário interceptar todas as instruções de um método. Aos blocos de código pretendidos, relativos a ciclos ou a construtores condicionais, está sempre associada uma ou mais condições que permitirão que uma nova iteração seja executada ou que as instruções referentes a um bloco de código sejam executadas. Este tipo de instrução é codificada juntamente com um valor com sinal (*offset*) que re-

[1] Trabalho parcialmente suportado pelo projecto PPC-VM (POSI / CHS / 47158 / 2002) financiado pela FCT e por fundos europeus (FEDER).

presenta o número de bytes a incrementar ao endereço actual, por forma a encontrar a instrução à qual o salto se refere. É a partir deste salto, considerando determinados padrões de codificação, que é possível concluir acerca do tipo de código fonte que está associado ao bloco de código analisado. Os padrões de codificação dos ciclos/construtores condicionais são normalmente definidos pelo sinal do *offset* e/ou pela instrução anterior à instrução destino do salto. Como resultado desta análise, obtém-se uma estrutura de dados hierárquica contendo todos os blocos de código acima referidos organizados hierarquicamente, juntamente com o número de instruções exclusivas de cada um.

Após identificar o tipo de bloco de código e a condição de salto associada, foi preciso concluir acerca da sua possível dependência em relação aos argumentos do método em questão. Para concretizar esta tarefa, foi necessário emular a pilha da JVM para se poder seguir a propagação das várias dependências de cada um dos argumentos, durante a execução do método. Esta fase obriga a tratar de forma distinta a pilha de dados e a pilha dos operandos. Qualquer instrução consome zero ou mais elementos da pilha de dados e pode produzir elementos nela. Deste modo chega-se a uma expressão de dependência em que cada um dos elementos produzidos depende da soma de todas as dependências dos elementos consumidos. As instruções em que é necessário considerar os operandos são as operações de `load/store`, incluindo aquelas que acedem a variáveis/campos de classe. No final, a informação resultante consiste num relatório que exibe todos os blocos de códigos importantes na decisão do controlo de fluxo do método, bem como a dependência das suas condições de salto para com os argumentos do método.

Pretende-se que a estimativa estática da complexidade apresente o mesmo padrão de comportamento que o tempo de execução medido dinamicamente. Deste modo, é esperado que quanto maior for a estimativa da complexidade para um determinado método, maior será o seu tempo de execução. Por forma a definir uma métrica fiável, foi desenvolvido um algoritmo (i) que contempla a recursividade no código e os saltos na linha de execução, causados por condições de salto ou ciclos e (ii) que penaliza a recursividade e os blocos mencionados. O valor da penalização é multiplicado pelo valor da complexidade previamente calculado para o bloco de código ao qual a penalização se refere. Basicamente, o algoritmo considera que todas as instruções demoram o mesmo tempo de processamento, excepto no caso de uma invocação, em que é calculado o valor da complexidade da transitividade associada ou no caso de haver uma penalização.

4 Resultados

Como exemplo, apresenta-se a classe `RayTracer` do benchmark `RayTracer` versão MPJ [6], ao qual foi aplicada a metodologia de cálculo da métrica de complexidade. O valor calculado estaticamente para a complexidade dos métodos da classe `RayTracer` é apresentado na tabela 1. Para validar a estimativa da complexidade, apresenta-se na mesma tabela o tempo de execução destes métodos, medido por uma ferramenta de instrumentação dinâmica desenvolvida no âmbito do projecto PPC-VM. Com a excepção de um caso, os métodos com uma complexidade maior apresentam um tempo de execução superior aos restantes.

Método	*Complexidade*	*Texec (ms)*
render	**7382**	**9397.0**
trace	**4563**	0.121966
shade	**4329**	**3.0**
createScene	**1430**	**19.0**
TransDir	162	0.001608
SpecularDirection	127	0.001383
Shadow	97	0.006936
setScene	95	0.0
intersect	87	0.005184
setHeight	4	0.0
setWidth	4	**2.0**

Tabela 1: Complexidade e tempo de execução dos métodos da classe `RayTracer`.

5 Conclusões

Ao optar pela análise de bytecodes, em detrimento de código fonte Java, o presente trabalho mostrou que a análise estática das aplicações fica facilitada. Outro contributo do presente trabalho foi o de solucionar a questão da fuga de referências, que não era tratada de forma adequada nos *profilers open-source* de Java. Os resultados obtidos com o exemplo RayTracer mostraram que a métrica de complexidade desenvolvida, calculada estaticamente, segue o mesmo padrão que o tempo de execução medido dinamicamente. Concluiu-se assim que, embora a estimativa da complexidade dos métodos não seja precisa em termos de valor absoluto, como possui uma boa fidelidade em relação ao tempo de execução, permite tomar decisões correctas aquando da distribuição da aplicação Java.

Referências

[1] João L. Sobral, António J. Esteves, João M. Fernandes, and Luís P. Santos. *Projecto PPC-VM: Parallel Computing Based on Virtual Machines.* Dep. Informática, Universidade do Minho, March 2004 - March 2007. `http://gec.di.uminho.pt/ppc-vm`.

[2] JeanTessier. *The Dependency Finder User Manual (Version 1.1.1).* Google Inc, Mountain View, California, 2005. `http://depfind.sourceforge.net/Manual.html`.

[3] Mike Atkinson. Jrefactory project, 2004. `http://jrefactory.sourceforge.net`.

[4] Diomidis Spinellis. *Chidamber and Kemerer Java Metrics (ckjm).* Department of Management Science and Technology, Athens University of Economics and Business, 2005. `http://www.spinellis.gr/sw/ckjm/doc/indexw.html`.

[5] André Spiegel. *Automatic Distribution of Object-Oriented Programs.* PhD thesis, FB Mathematik und Informatik, Frei Universitat Berlin, July 2002.

[6] *Java Grande Benchmarking Project.* Edinburgh Parallel Computing Centre, University of Edinburgh, 2005. `http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/`.