2
Referencial Teórico

Um programa Java é escrito em um arquivo `java`. Um compilador (em geral o JavaC) é então usado para gerar os bytecodes e armazená-los em um arquivo *class*. Este arquivo é composto de várias partes: um cabeçalho de 8 bytes, seguido da tabela de constantes (*constant pool*) onde números, *strings* e identificadores são armazenados; uma parte com informações sobre a classe, outra com a lista dos campos (*fields)* da classe e por fim uma parte com os métodos da classe.

Os bytecodes da linguagem Java foram definidos de forma a serem compactos e a facilitarem o trabalho do verificador de bytecodes. O verificador é executado na plataforma-alvo, antes de iniciar a interpretação dos bytecodes. Ele é usado para garantir que o código foi gerado por um compilador confiável e que não foi alterado após a compilação, de forma que possa ser executado sem quebras de semântica da linguagem.

Os bytecodes podem ser divididos nas categorias de operações aritméticas e lógicas, conversão entre tipos primitivos, movimentação de dados de e para a pilha, desvios condicionais ou diretos, alocação de memória, e outros, detalhados na tabela 1. A tabela 33 do apêndice lista os bytecodes, dispostos em ordem numérica de acordo com a especificação definida pelo JCP.

Nem todas as operações estão disponíveis para todos os tipos, que estão listados na tabela 2. Por exemplo, não existe bytecode para somar dois valores do tipo char; eles devem ser convertidos para inteiro, somados, e então convertidos de volta para char. A regra geral é que tipos menores que o inteiro devem ser convertidos para inteiro, operados, e então o resultado convertido de volta para o tipo original.

Os bytecodes Java e o formato do arquivo *class* estão longe da perfeição. Diversos artigos sugerem modificações e melhorias em ambos, tanto para tornar mais rápida a execução quanto para deixar o código mais compacto.

Alguns artigos se baseiam na identificação das seqüências de bytecodes mais freqüentes e a criação de novos bytecodes que os substituam, diminuindo o tamanho dos programas e melhorando assim o desempenho. Outros, na alteração do formato da tabela de constantes.

Categoria	Operações
Operações aritméticas	soma, subtração, multiplicação, divisão e resto, negação
Operações lógicas	deslocamento à esquerda, deslocamento à direita sem sinal e com sinal, e, ou, ou-exclusivo
Conversão entre tipos primitivos	int para char/byte/short/long/float/double, long para int/float/double, float para int/long/double, double para int/long/float
Movimentação entre dados e a pilha	cerca de 95 bytecodes.
Desvios	condicional, direto, sub-rotina, retorno de sub-rotina, chamadas a métodos (da superclasse, da classe, estáticos e de interface)
Alocação de memória	objetos, vetores, matrizes, vetor de caracteres
Outros operadores	comprimento de vetor, verificação de pertinência de um objeto a uma classe, disparo de exceção, sincronização de tarefas, ponto de parada para depuradores

Tabela 1: Categorias dos bytecodes Java

Tipo	Precisão
char	inteiro de 16 bits sem sinal.
byte	inteiro de 8 bits com sinal.
short	inteiro de 16 bits com sinal.
int	inteiro de 32 bits com sinal.
long	inteiro de 64 bits com sinal.
float	ponto flutuante de 32 bits.
double	ponto flutuante de 64 bits.
boolean	valores verdadeiro ou falso.
Object	objeto composto de outros objetos e tipos primitivos

Tabela 2: Tipos Java

As análises de freqüência podem ser estáticas ou dinâmicas. As estáticas analisam apenas o arquivo `class`, enquanto que as dinâmicas analisam a execução de um programa. Cada uma serve para um propósito. A dinâmica permite conhecer as partes mais requisitadas de um programa, e dessa forma melhorar a eficiência deste programa. Todavia, sua análise pode ser tendenciosa na medida em que partes do programa poderão não ser analisadas, caso não sejam executadas. Já a estática permite analisar o programa em sua totalidade, mas

não leva em consideração se essas partes do programa serão efetivamente usadas. Em geral, a análise estática é usada em bibliotecas de classes, enquanto que as dinâmicas são mais indicadas para a análise do fluxo de um programa. Dowling et al. (2001) afirma que, salvo poucas exceções, não existe correlação linear entre análises estáticas e dinâmicas de bytecodes. Portanto, resultados de testes estáticos e dinâmicos não podem ser comparados diretamente.

2.1. Tamanho do código armazenado

Antonioli & Pilz (1998) foi base de muitos dos artigos que analisaram aspectos do arquivo `class`. Através da análise estática de cerca de 4 mil classes, ele descobriu que a tabela de constantes ocupa a maior parte do arquivo `class`, enquanto que os bytecodes que formam o programa consumiram uma parte muito pequena (figura 1). Analisando a tabela de constantes, a maior parte é gasta com identificadores de classes, campos e métodos. Conseqüentemente, se dividirmos a tabela de constantes em áreas distintas, poderemos economizar uma parte significativa, ao eliminarmos informações que servem para definir o tipo de identificador armazenado (figura 2).

Pugh (1999) sugere formas de se melhorar a compactação dos arquivos JAR. Estes arquivos são usados para comprimir classes Java, usando o algoritmo de Lempel-Ziv, onde cada classe é compactada de forma individual. O autor informa que uma melhor compactação é obtida se todas as classes forem concatenadas e compactadas como um só arquivo[4]. Sabendo que a tabela de constantes ocupa a maior parte de um arquivo class (Antolioli & Pilz, 1998), ele sugere que várias classes compartilhem uma mesma tabela de constantes. Além disso, ao agrupar as entradas da tabela de constantes pelo tipo de identificador armazenado (método, campo, classe, etc), é possível utilizar um índice de 8 bits ao invés dos usuais 16 bits. Outras sugestões para compactação são apresentadas no artigo, porém as descritas aqui foram as mais importantes.

Rayside et al. (1999) também discorre sobre técnicas para reduzir o tamanho do código gerado pelo compilador Java. Ele propõe que a descrição de um método, com seus parâmetros e tipo de retorno, seja quebrado em um vetor de índices para a tabela de constantes, ao invés da forma usual, que é colocar todas essas informações em uma única string. Entre as técnicas propostas, esta é

[4] Esse formato de arquivo é conhecido como *arquivo sólido*, e é empregado por diversos programas de compactação, como o 7Zip (Pavlov 1999)

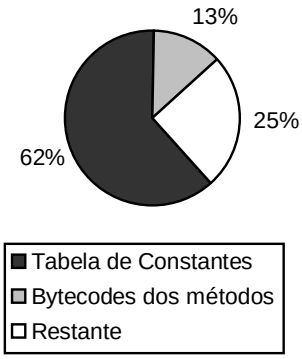

Figura 1: Composição de um arquivo class

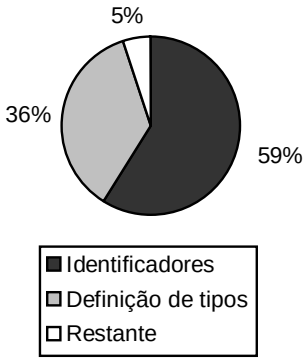

Figura 2: Tabela de constantes

a que proporciona o maior ganho de compactação. O autor também sugere a separação dos bytecodes de seus operandos, para gerar seqüências mais homogêneas e assim melhorar não apenas a compactação, mas também a reorganização da tabela de constantes, como em Pugh (1999). Ele afirma ainda que o compartilhamento da tabela de constantes entre um conjunto de classes resulta em arquivos JAR 50% menores, na média.

2.2. Desempenho dos programas

Um estudo sobre freqüências no uso de bytecodes que serve como base para muitos artigos é Daly et al. (2001). Nele são usados cinco programas de benchmark para coletar a freqüência dinâmica do código gerado por diversos compiladores, além do tamanho da lista de variáveis locais e de operandos usados nos métodos. Uma análise interessante foi feita através da separação das instruções em grupos, e a coleta da média das ocorrências. Na tabela 3, onde são exibidos os cinco primeiros colocados, vemos que quatro deles são usados para carregar dados para a pilha, representando quase dois terços das instruções. Com isso é possível avaliar o quanto uma máquina de registradores economizaria, pois a maioria dessas cargas seria eliminada. O artigo termina comentando a baixa quantidade de otimizações feitas pelos compiladores, dificultando o trabalho da máquina virtual.

As análises de Antonioli & Pilz (1998) e Daly et al. (2001) são feitas com apenas 1 bytecode. Donoghue et al. (2002), por sua vez, examina a freqüência dinâmica dos *pares* de bytecodes mais usados. Doze programas de *benchmarks* populares são analisados. Os testes realizados usam apenas 152 bytecodes dos 202 disponíveis. O resultado mais importante é a constatação que a seqüência

`aload_0 getfield`, que coloca no topo da pilha o valor de um campo de instância, aparece com 9,21%. Para se ter uma idéia, a soma das próximas 5 freqüências não atinge este valor. Das 10 maiores freqüências, `aload_0` aparece em 4 delas, e `getfield` em 5.

Outro estudo sobre seqüências de bytecodes é Stephenson & Holst (2004). Nele, ao invés de 1 ou 2 bytecodes, são analisadas seqüências de até 20 bytecodes. Ao todo treze programas são testados, e apenas 162 bytecodes são usados. Uma das conclusões é que doze dos treze programas testados tinham, no topo da lista de ocorrências, seqüências de bytecodes usados para incrementar o valor de um campo. Essas instruções estão descritas na tabela 4.

Estes dois estudos, Donoghue et al. (2002) e Stephenson & Holst (2004) indicam que certas alterações nos bytecodes, tais como incrementar um campo com menos instruções, podem facilitar o acesso e a operação dos campos de uma classe, otimizando assim o seu desempenho.

Donoghue & Power (2004), dando seguimento ao seu estudo inicial (Donoghue et al., 2002), analisam seqüências de bytecodes, e exibem alguns resultados para as de tamanho 2, 4, 8, 16 e 32. Como em outros estudos, `aload_0`

Grupo	Freqüência
Carga de locais	35,9
Campos de objetos	16,5
Aritméticos	13,0
Carga de vetor	7,1
Carga de constantes	5,8

Tabela 3: Percentagem da freqüência dinâmica para grupos de Bytecodes

Bytecode	Descrição	Simulação da Pilha
`aload_0`	Carrega para o topo da pilha a instância da classe	`this`
`dup`	Duplica o valor presente no topo	`this this`
`getfield`	Carrega para o topo da pilha o valor do campo	`this valor_campo`
`iconst_1`	Coloca no topo da pilha o valor 1	`this valor_campo 1`
`iadd`	Adiciona os dois valores presentes no topo da pilha	`this (valor_campo+1)`
`putfield`	Coloca o resultado no campo	`<vazia>`

Tabela 4: Bytecodes necessários para somar 1 a um campo de instância
Na simulação da pilha, o topo está à direita.

`getfield` aparece no topo dos resultados para seqüências de tamanho 2. A seqüência de bytecodes que incrementa em 1 o valor de um campo também aparece no topo das seqüências de tamanhos 8 e 4, sendo que nesta falta apenas a última instrução, `putfield`.

A seguir, examinaremos brevemente as máquinas de registradores, e suas vantagens perante as máquinas baseadas em pilha.

2.3. Máquinas virtuais baseadas em registradores

Em máquinas virtuais baseadas em registradores, também chamadas de *Register-Transfer Machine* (Craig, 2005), registradores são usados para armazenar os operandos. Máquinas deste tipo têm um bom número de registradores. Como em geral este número é maior que a quantidade de registradores do processador destino, é preciso armazená-los em uma estrutura, como um vetor, por exemplo. Visto que uma máquina virtual baseada em pilha também armazena a pilha em um vetor, a princípio não haveria vantagem no uso da máquina baseada em registradores. Há, porém, pelo menos duas vantagens: primeiro, não há necessidade de se incrementar e decrementar o ponteiro de topo da pilha; segundo, após carregar a local para um registrador e ela não for sobrescrita, não é preciso recarregá-la caso seja requisitada novamente (Craig, 2005). Além disso, diversas otimizações podem ser aplicadas por um compilador, como a passagem de parâmetros entre rotinas em registradores, ao invés de se usar a pilha.

Por outro lado, as maquinas baseadas em registradores tendem a ser mais complexas, pois os operandos devem ser especificados, o que não ocorre nas baseadas em pilha, onde eles estão sempre no topo da pilha. Além disso, é também exigido um esforço maior do compilador na geração do código.

O custo de se executar uma instrução em uma máquina virtual baseada em pilhas ou registradores pode ser calculado através de três componentes (Ertl, et al., 2005):

- Desvio para a instrução
- Acesso aos operandos
- Execução da instrução

O desvio para a instrução (*instruction dispatching)* pode ser feito através de duas formas básicas: `switch/case` e desvio direto. O `switch/case`, muito usado nos interpretadores, tem dois sérios problemas. Em primeiro lugar, os compiladores inserem um teste para verificar se o operando do `switch` está

dentro dos limites, mas isso é desnecessário no caso dos interpretadores. O segundo problema decorre da arquitetura dos processadores modernos, onde existe a predição de desvios: como no `switch` existe apenas um local que faz o desvio para todas as instruções, o índice de falhas na predição varia de 80% a 98% (Ertl et al., 2001).

A segunda forma é através de desvio direto (*threaded code*), onde os endereços das instruções são guardados em um vetor e o `switch` é então substituído por um `goto` em cada instrução. As vantagens são a eliminação dos testes de limite e a drástica redução no índice de falhas de predição de desvio. Cada instrução tende a desviar para um número pequeno de instruções, o que aumenta o índice de acertos para algo em torno de 55% (Ertl et al., 2001). O único problema é que poucos compiladores, como o GCC, suportam o desvio direto, pois a sintaxe não faz parte do ANSI/C.

O acesso aos operandos é responsável pela segunda parte no custo para executar uma instrução. Neste ponto, as máquinas baseadas em registradores perdem um pouco para as baseadas em pilha, porque os operandos envolvidos primeiro devem ser localizados, para que seus valores sejam inseridos nos registradores. Vale ressaltar que o custo de acesso aos operandos é bem menor que o custo da falha nas predições de desvio.

O último ponto, a execução da instrução, é equivalente nos dois tipos de máquinas virtuais. Porém, o uso de registradores pode beneficiar-se de algum ganho de otimização, como o reaproveitamento de valores guardados em registradores, pois o uso dos operandos não é destrutivo como em máquinas baseadas em pilha.

2.4. Conversão de pilha para registradores

Alguns trabalhos abordaram a conversão de bytecodes Java para instruções de manipulação de registradores. Em um destes estudos (Davis et al., 2003), o número de instruções caiu cerca de 35%, enquanto que o tamanho do código aumentou cerca de 45%, devido à necessidade de se especificar os operandos. A seguir, destacamos suas principais conclusões:

- Variáveis locais e da pilha são diretamente convertidas para registradores. Como essa tradução gera um grande número de instruções do tipo “imove r6,r10; imove r10,r6”, foi necessário implementar o algoritmo de propagação de cópia (*copy propagation*) no conversor. Duas versões deste algoritmo foram testadas: uma simples, para blocos básicos, e outra complexa, para o método inteiro; esta última melhorou o desempenho do código apenas 1% a

mais que a simples. Além dessa otimização, a de remoção de código morto (*dead code elimination*) também foi aplicada. Ambas resultaram em uma diminuição de cerca de 28% nas instruções de movimentação.

- A chamada de métodos foi implementada de duas formas. Na primeira versão, as instruções de movimentação antecediam a chamada do método, de forma a empilhar os parâmetros. Em uma segunda versão, os índices dos registradores eram passados como parâmetros para a instrução. Apesar de gerar um aumento no número de bytes na chamada da instrução, a estratégia provou ser mais eficiente porque permitiu a retirada de várias instruções precedentes (valendo-se da regra que o custo da leitura destes parâmetros é menor que o da inserção de novas instruções).

O trabalho de Ertl et al. (2005), ampliando o de Davis et al. (2003), adotou estratégias mais agressivas para a diminuição do tamanho e quantidade do código gerado. Ele conseguiu reduzir o número de bytecodes em 47%, enquanto que o tamanho do código cresceu apenas 25%. Além disso, ele obteve uma redução no tempo de execução do programa de 26%, quando comparado à máquina de pilha.

As conversões utilizadas em Ertl et al. (2005) foram:

- As instruções *pop* e *pop2,* que retiram dados da pilha, foram eliminadas.
- As instruções de *load* e *store* foram convertidas em *move*.
- As instruções para manipulação de pilha, como *dup*, *dup2*, *dupx2*, etc., foram convertidas em instruções *move* correspondentes.
- A otimização de *copy propagation* (para frente e para trás) foi bastante aplicada.
- Como forma de otimizar as instruções que manipulam constantes, todas são movidas para registradores no início do método.

Boa parte das conclusões apresentadas nos artigos descritos acima foi adotada em nossa nova arquitetura, que será descrita no próximo capítulo.