PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Guilherme Campos Hazan

Uma Especificação de Máquina de Registradores para Java

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Roberto Ierusalimschy

PUC-Rio, março de 2007

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Guilherme Campos Hazan

Uma Especificação de Máquina de Registradores para Java

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática – PUC-Rio

Profa. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática – PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de janeiro, 09 de abril de 2007

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Guilherme Campos Hazan

Bacharel em Ciência da Computação pela Universidade Federal do Rio de Janeiro (UFRJ). Começou a programar em 1984. Iniciou em Java em 1998, trabalhando para a Quality Software, onde criou applets de geração de diversos tipos de gráficos, além de uma applet de CAD (2o lugar no Concurso Anual de Comércio Eletrônico da Siemens de 2002). Palestrante em diversas conferências: Comdex Sucesu 2002, One Day Java 2002, Just Java 2003, Java In Rio 2003, Congresso Catarinense de Software Livre 2003, dentre outros. Guilherme é o criador e da máquina virtual SuperWaba. Professor de Java para PDAs na pós-graduação da CESUMAR.

Ficha Catalográfica

Hazan, Guilherme Campos

Uma especificação de máquina de registradores para Java / Guilherme Campos Hazan; orientador: Roberto Ierusalimschy. – 2007.
72 f. ; 30 cm

Dissertação (Mestrado em Informática)–Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.
Inclui bibliografia

1. Informática – Teses. 2. Java. 3. Bytecode. 4. Compilador. 5. Máquina virtual. 6. Registrador. 7. Dispositivos móveis. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.
Incluí referências bibliográficas.

Java, bytecode, compilador, máquina virtual, registrador, pilha, desempenho, otimização, pda, dispositivos móveis

CDD: 004

Gostaria de dedicar essa dissertação à Verinha, minha mãe, que faleceu em Fevereiro de 2005. Ela, como Diretora da Escola de Musica da Universidade Federal de Minas Gerais, e Diretora Artística da Fundação Clovis Salgado, foi um exemplo para mim, pelo seu profissionalismo reconhecido pelos colegas de trabalho e capacidade de tornar um sucesso os trabalhos que fez pela grande dedicação e, principalmente, pela paixão com que tratava a sua profissão.

Agradecimentos

Gostaria de agradecer ao meu orientador, Roberto, pelo seu desprendimento em me orientar em um trabalho diferente do que costuma realizar, na plataforma Lua. Isso causava surpresa em todas as pessoas com quem eu conversava sobre o assunto da dissertação e sobre quem era o orientador. Elas indagavam: "... mas nao é sobre Lua?".

Gostaria de agradecer também ao Prof. Arndt von Staa, cuja matéria sobre Testes e Métricas de Software foi de fundamental importância para auxiliar a correta implementação da máquina virtual.

Resumo

Hazan, Guilherme Campos; Ierusalimschy, Roberto. **Uma Especificação de Máquina de Registradores para Java.** PUC-Rio, 2007. 72p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A linguagem Java foi definida tendo como foco a portabilidade. O código gerado pela compilação é interpretado por uma máquina virtual, e não diretamente pelo processador destino, como um programa em C. Este código intermediário, também conhecido como bytecode, é a chave da portabilidade de Java. Os Bytecodes Java usam uma pilha para manipular os operandos das instruções. O uso de pilha tem suas vantagens e desvantagens. Dentre as vantagens, podemos citar a simplicidade da implementação do compilador e da máquina virtual. A principal desvantagem é a redução na velocidade de execução dos programas, devido à necessidade de se mover os operandos para a pilha e retirar dela o resultado, gerando um aumento no número de instruções que devem ser processadas. Diversos estudos indicam que máquinas virtuais baseadas em registradores podem ser mais rápidas que as baseadas em pilha. Decidimos criar uma nova especificação de bytecodes, específicos para máquinas virtuais baseadas em registradores. Esperamos com isso obter um aumento no desempenho das aplicações.

Palavras-chave

Java; bytecode; compilador; máquina virtual; registrador; pilha; desempenho; otimização; pda; dispositivos móveis

Abstract

Hazan, Guilherme Campos; Ierusalimschy, Roberto. **A Specification for a Java Register-Based Machine.** PUC-Rio, 2007. 72p. MSc Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The Java language was created with a focus on portability. The code generated by the compiler is interpreted by a virtual machine, and not directly by the target processor, like programs written in C. This intermediate code, also known as bytecode, is the key to Java's portability. The Java Bytecodes use a stack to manipulate the instruction operands. The use of stack has its their pros and cons. Among the advantages, we can cite the simplicity of implementation of the compiler and virtual machine. On the other hand, there is a speed reduction in the program's execution, due to the need to move the operands to and from the stack, and retrieve results from it, increasing the number of instructions that are processed. Much study has been done that indicating that register-based virtual machines can be faster than the ones based on stacks. Based on this, we decided to create a new bytecode specification, proper for a virtual machine based on registers. By doing this, we hope to obtain an increase in an application's performance.

Keywords

Java; bytecode; compiler; virtual machine; register; stack; performance; optimization; pda; mobile devices

Sumário

1 Introdução 13

2 Referencial Teórico 15
2.1. Tamanho do código armazenado 17
2.2. Desempenho dos programas 18
2.3. Máquinas virtuais baseadas em registradores 20
2.4. Conversão de pilha para registradores 21

3 Nova Arquitetura de Bytecodes 23
3.1. Separação dos registradores em grupos 23
3.2. Quantidade de registradores 24
3.3. Operações com campos 25
3.4. Ortogonalidade das operações 25
3.5. Otimização da tabela de constantes 27
3.6. Arquitetura de palavra das instruções 27
3.7. Formato das instruções 28
3.8. Escolha das instruções 28
3.9. Listagem das instruções 37

4 Avaliação do Código Gerado 46
4.1. Descrição dos testes 46
4.2. Desempenho obtido nas plataformas 50
4.3. Tamanho do código gerado 54

5 Conclusão 56

6 Referências Bibliográficas 59

7 Apêndice 61
7.1. Lista dos bytecodes Java 62
7.2. Mapeamento entre os bytecodes Java e as novas instruções 63

Lista de tabelas

Tabela 1: Categorias dos bytecodes Java 16
Tabela 2: Tipos Java 16
Tabela 3: Percentagem da freqüência dinâmica para grupos de Bytecodes 19
Tabela 4: Bytecodes necessários para somar 1 a um campo de instância 19
Tabela 5: Percentagem do número de parâmetros usados em métodos 24
Tabela 6: Percentagem do número de locais usados em métodos 24
Tabela 7: Programas usados durante o teste e resumo dos resultados 32
Tabela 8: Instruções aritméticas e lógicas dos programas Java 33
Tabela 9: Instruções aritméticas e lógicas do J2SE 5.0 33
Tabela 10: Operandos das instruções aritméticas e lógicas para Java 34
Tabela 11: Operandos das instruções aritméticas e lógicas para J2SE 5.0 34
Tabela 12: Instruções de desvio condicional para Java 35
Tabela 13: Instruções de desvio condicional para J2SE 5.0 36
Tabela 14: Operandos das instruções de desvio condicional para Java 36
Tabela 15: Operandos das instruções de desvio condicional para J2SE 5.0 37
Tabela 16: Número de locais por método para programas Java 37
Tabela 17: Número de locais por método para J2SE 5.0 37
Tabela 18: Operandos usados nas instruções 40
Tabela 19: Instruções de movimentação 41
Tabela 20: Instruções aritméticas e lógicas 42
Tabela 21: Instruções de desvio condicional 43
Tabela 22: Instruções de chamada de método 44
Tabela 23: Instruções de conversão entre tipos de dados 44
Tabela 24: Instruções para retorno de método 44
Tabela 25: Instruções não categorizadas 45
Tabela 26: Partes que diferem entre as plataformas 48
Tabela 27: Partes comuns às plataformas 48
Tabela 28: Código fonte dos testes realizados no Pentium 4-M. 49
Tabela 29: Tempos em milisegundos dos testes realizados no Pentium 4-M 53
Tabela 30: Número de instruções dentro do laço 53
Tabela 31: Tempos em milisegundos dos testes realizados no Dell Axim X3 53
Tabela 32: Bytecodes Java 62

Tabela 33: Conversão entre bytecodes Java e as novas instruções 66

Lista de figuras

Figura 1: Composição de um arquivo class 18
Figura 2: Tabela de constantes 18
Figura 3: Comparação dos tipos primitivos em relação ao `int`. 26
Figura 4: Código usado para medir o tempo de execução 26
Figura 5: Gráfico comparativo com os tempos obtidos no Pentium 4-M 52
Figura 6: Gráfico comparativo com os tempos obtidos no Dell Axim X3 52
Figura 7: Tamanho do código gerado pela compilação do teste 55
Figura 8: Composição do arquivo gerado pela compilação do teste 55
Figura 9: Tamanho do código gerado pela compilação dos protótipos 55

1
Introdução

A máquina virtual Java é bastante difundida nos dias de hoje. Ela está presente em servidores, computadores de mesa, navegadores da web, e dispositivos móveis como celulares. Ela é a chave para a portabilidade da linguagem: o compilador Java gera um código intermediário (doravante denominado *bytecode* Java), independente de plataforma ou sistema operacional, que é interpretado pela máquina virtual.

A linguagem Java, criada pela Sun Microsystems, é atualmente controlada por um comitê gestor, denominado JCP (*Java Community Process*). Isso faz com que novas funcionalidades tenham que passar por um crivo para serem incorporadas à linguagem. Na época de criação do Java, o JCP definiu que a máquina virtual seria baseada na manipulação dos operandos através de uma pilha. Um fator determinante da escolha foi a maior simplicidade para a implementação do compilador e da máquina virtual baseada em pilha, comparativamente ao modo tradicional usado nos processadores, que é a manipulação de operandos através de registradores.

Diversos estudos (Davis et al. 2003; Ertl et al., 2005) indicam que máquinas virtuais baseadas em registradores podem ser mais rápidas que as baseadas em pilha. Para exemplificar, vejamos o código abaixo de pseudo-instruções para a expressão “a = b + c”, em uma máquina virtual baseada em pilhas (coluna da esquerda) e outra baseada em registradores (coluna da direita):

```
iload b          iadd a, b, c
iload c
iadd
istore a
```

A máquina virtual baseada em pilha gasta mais instruções, pois requer que cada variável seja colocada na pilha, que é de onde a instrução *iadd* irá retirar os operandos para efetuar a soma. O resultado é então guardado de volta na pilha e então armazenado na variável destino. Já na baseada em registradores não é preciso mover os operandos para nenhum local temporário, e consegue-se fazer o mesmo com menos instruções.

Visto que o desempenho de Java era bem inferior ao das linguagens tradicionais como C, uma solução foi adotada: a criação dos compiladores em tempo real (JIT - *Just In-Time compiler*). Os compiladores JIT transformam os bytecodes Java em código de máquina do processador-alvo no momento da execução; com isso consegue-se um ganho de desempenho na ordem de 10 vezes. Porém, a criação destes compiladores é difícil e deve ser feita para cada processador-destino. Além disso, o JIT aumenta o uso de memória, e também faz com que ocorra um atraso na execução do programa, pois na primeira vez que o método ou classe é carregado, devem ser compilados e convertidos para código de máquina.

Os computadores móveis atuais, conhecidos como assistentes pessoais digitais (*PDA*, da sigla em inglês), têm poder de processamento cerca de 10 a 30 vezes menor que os computadores de mesa[1], e memória dezenas de vezes menor[2]. Nesses equipamentos, o atraso inicial provocado por um JIT é bastante perceptível.

Sendo o nosso foco equipamentos móveis, como PDAs e *Smartphones*[3], decidimos criar uma nova especificação de bytecodes, específicos para máquinas virtuais baseadas em registradores. Esperamos com isso obter um aumento no desempenho das aplicações nestes dispositivos, sem requerer o uso de compiladores JIT.

Tendo sempre o cuidado de minimizar os requisitos de memória da nova especificação, iremos apresentar, no capítulo 2, resumos de artigos que sugerem modificações em aspectos da linguagem que permitam uma redução no tamanho do código gerado, assim como um aumento na eficiência do mesmo. Em seguida, no capítulo 3, iremos apresentar as principais características da nova especificação, além de descrever os resultados obtidos por um analisador estático de código Java que usamos para validar a escolha das instruções. No capítulo 4, mostraremos resultados de código gerado com a nova arquitetura, e, finalmente, no capítulo 5 apresentaremos as conclusões relativas ao trabalho.

---

[1] Muitos computadores de mesa atuais têm um *clock* de 3 GHz, enquanto que os PDAs mais comuns têm de 100 a 300 MHz.

[2] Os computadores de mesa possuem entre 512MB e 2GB mais o armazenamento em disco rígido (80GB), enquanto que os PDAs possuem entre 32MB e 128MB, que devem ser compartilhados entre memória RAM e armazenamento de programas.

[3] Celulares com uma maior capacidade de processamento.

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 u-sadas. 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)

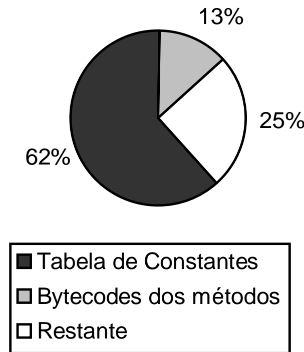

Figura 1: Composição de um arquivo class

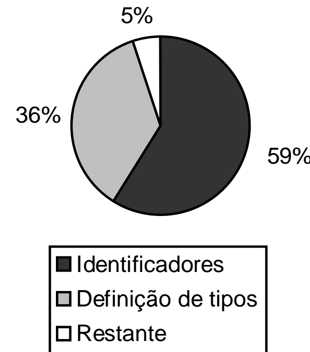

Figura 2: Tabela de constantes

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.

3
Nova Arquitetura de Bytecodes

No capítulo anterior procuramos descrever as principais propostas de alterações na arquitetura da linguagem Java, apresentadas por diversos autores. Nesse capítulo, da seção 3.1 a 3.7, enunciaremos as premissas com as quais a especificação foi projetada. Na seção 3.8 iremos descrever um interpretador virtual que foi usado para ajudar na definição das instruções da nova arquitetura, que serão por fim descritas na seção 3.9.

3.1.
Separação dos registradores em grupos

Em Java, as variáveis locais e os parâmetros de um método são armazenados em uma estrutura na memória, manipulada como uma pilha. Esta estrutura armazena tipos de dados diferentes (`int`, `long`, `double`, `Object`) e ao mesmo tempo, de tamanhos distintos: `int` tem 32 bits, `Object` depende do processador pois geralmente é armazenado como um ponteiro, `long` e `double` têm 64 bits (Davis et al., 2003; Hsieh et al., 1996). Tipos de 64 bits podem ser armazenados em duas posições consecutivas, caso uma estrutura de 32 bits seja escolhida. A mistura de tipos na pilha dificulta o trabalho do coletor de lixo, pois torna necessário que se determine, durante a execução do coletor, se o tipo na pilha é um objeto, passível de ser coletado.

Para facilitar o trabalho do coletor de lixo, o manuseio dos tipos primitivos e a definição das instruções, decidimos pela separação dos registradores em três estruturas diferentes: uma para tipos `int` (32 bits), outra para tipos `double` e `long` (64 bits), e uma terceira para objetos. A estrutura própria para objetos irá acelerar o coletor de lixo, pois uma parte substancial de seu processamento é gasta para determinar se o que está sendo varrido na estrutura é um objeto ou um tipo primitivo. No nosso caso, bastará varrer a pilha de objetos para descobrir os que estão em uso.

Inicialmente pensamos em separar os tipos primitivos em três estruturas, uma para int, outra para double, e uma terceira para long. Porém, quando fize-

mos a implementação vimos uma otimização de desempenho juntando as estruturas de double e long em uma só, para tipos de 64 bits..

3.2.
Quantidade de registradores

Em nossa máquina virtual, as variáveis locais e os operandos dos métodos invocados serão armazenados em registradores. Definimos que o número máximo de registradores usados em cada método será de 256, sendo 64 para cada um dos tipos `int`, `Object`, `double` e `long`.

Para certificar que este valor atende às aplicações Java, nos apoiamos em um artigo onde é feita uma análise da distribuição do número de parâmetros passados em um método e também do número de variáveis locais criados para o método (Waldron, 1999). Os resultados obtidos através da análise dos programas JAS (Java Assembler) e o JCC (Java Compiler Compiler) mostram que a quase totalidade dos métodos usam no máximo 9 locais e 9 parâmetros (tabelas 5 e 6), resultando em um total de 18 variáveis. O artigo falha, porém, ao não informar o número máximo de variáveis obtido nos testes realizados.

Considerando-se que uma máquina virtual baseada em registradores conterá, em média, um número de registradores que seria a soma do número de parâmetros com o número de variáveis locais, conclui-se com base nas tabelas 5 e 6, que a quase totalidade dos métodos teria um tamanho de no máximo 20 registradores. Assim sendo, estipular um máximo de 64 registradores para cada um dos tipos atende à imensa maioria dos aplicativos Java. Para eventuais programas onde esse número seja extrapolado, um erro de compilação deverá ser gerado, obrigando o usuário a reescrever o método em questão.

#	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**> 9**
jas	6,0	19,2	19,4	34,2	9,2	9,9	1,2	0,0	0,9	0,0	0,0
jcc	5,9	11,5	24,4	13,7	20,3	16,5	3,5	3,4	0,3	0,3	0,1

Tabela 5: Percentagem do número de parâmetros usados em métodos

#	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**> 9**
jas	0,2	42,9	24,6	9,6	6,0	7,8	7,2	0,6	0,9	0,0	0,0
jcc	10,4	27,2	30,4	8,4	8,0	9,2	1,4	1,6	3,0	0,1	0,3

Tabela 6: Percentagem do número de locais usados em métodos

3.3. Operações com campos

Alguns artigos da bibliografia apresentam sugestões para a criação de novos bytecodes baseados na análise estática e dinâmica de bibliotecas e programas (Donoghue et al., 2002 e 2004; Stephenson & Holst, 2004; Waldron et al., 2000). Nas análises destes artigos, dois conjuntos de instruções sobressaem: a carga de operandos e de resultados para a pilha, e o acesso a campos. Um dos trabalhos sugere que 40% dos bytecodes são dedicados à carga de dados, que serão eliminadas com o uso dos registradores (Waldron et al., 2000).

Decidimos então estudar a criação de instruções que facilitem operações em campos de instância, conforme foi expressamente sugerido em Stephenson & Holst (2004).

3.4. Ortogonalidade das operações

Alguns tipos de Java, como `byte`, `short` e `float`, são bem menos usados que os demais (Waldron et al., 1999). Em geral, esses tipos são usados para a economia de memória no uso de vetores. Além disso, a linguagem Java especifica que as operações são não ortogonais, ou seja, as mesmas operações não se aplicam a todos os tipos de dados. Por exemplo, os tipos `byte`, `short` e `char` nunca são operados diretamente: são convertidos para `int`, operados, e convertidos de volta. Em vista destes resultados, decidimos manter a não ortogonalidade, dando preferência à definição de instruções para os tipos mais usados, `int`, `double` e `long`.

Em uma proposta mais radical, decidimos eliminar o tipo `float`. Em nossa plataforma alvo, os PDAs, o tipo `float` é pouco usado quando comparado com `double` (em geral usa-se esse tipo para armazenamento de valores monetários, que requerem uma maior precisão). As operações com `float` serão transformadas em operações com `double`. Grande parte dos processadores atuais possui um co-processador matemático, o que diminui bastante a diferença de desempenho entre esses tipos. Fizemos um teste, cujo código está na figura 4, para mensurar o custo das operações de ponto flutuante na linguagem C em dois PDAs de plataformas diferentes, mas com processadores equivalentes. A título de curiosidade, medimos também o desempenho do tipo `long`. O resultado

(figura 3) foi que o `double` é no mínimo 1,65 vezes (no caso do Palm OS) e no máximo 2,65 vezes (no Pocket PC) mais lento que o `float`. Este resultado demonstra que o uso de `double` ao invés de `float` não gera uma perda de desempenho tão grande quanto se supõe. O tipo `long` do Java (64 bits) demonstrou ser muito mais lento que o `int`, e cabe recomendar que seu uso seja evitado até que os processadores de 64 bits se tornem populares em PDAs.

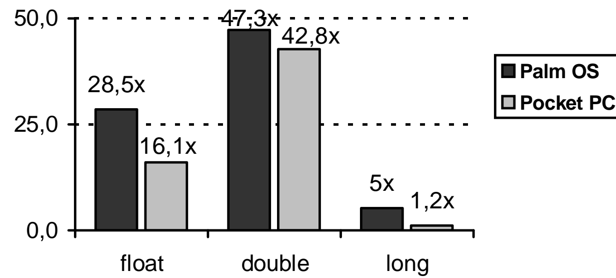

Figura 3: Comparação dos tipos primitivos em relação ao `int`.
Palm OS: Treo 650 com processador PXA270 XScale 312MHz.
Pocket PC: Dell Axim X3 com processador PXA270 XScale 300MHz.

```
static uint32 getTimeStamp()
{
#ifdef PALMOS
    return TimGetTicks() * 1000 / SysTicksPerSecond();
#else
   return GetTickCount();
#endif
}

static void debug(char* str, int32 t, int32 finalRes)
{
   printf(%s: %d\nfinal result (truncated): %d\n”,str,t,finalRes);
}

void teste()
{
   int32 i,ini;
   int32  ii=12345;     int64  ll=12345;
   float  ff=123.34f;   double dd=12334.456;

   ini = getTimeStamp();
   for (i = 10000000; i >= 0; i--) ii = 13 - ii;
   debug("int test", getTimeStamp()-ini,ii);
   ini = getTimeStamp();
   for (i = 10000000; i >= 0; i--) ll = 13 - ll;
   debug("long test", getTimeStamp()-ini,(int32)ll);
   ini = getTimeStamp();
   for (i = 10000000; i >= 0; i--) ff = 13 - ff;
   debug("float test", getTimeStamp()-ini,(int32)ff);
   ini = getTimeStamp();
   for (i = 10000000; i >= 0; i--) dd = 13 - dd;
   debug("double test", getTimeStamp()-ini,(int32)dd);
}
```

Figura 4: Código usado para medir o tempo de execução

3.5. Otimização da tabela de constantes

A tabela de constantes é uma área onde ficam armazenadas as definições textuais de classes, métodos e campos, além de constantes numéricas e strings. No caso do JDK, ela é responsável por até 65% do tamanho total das classes (Rayside et al., 1999).

Decidimos que, opcionalmente, a tabela de constantes poderá ser compartilhada por um determinado conjunto de classes, possivelmente gerando reduções de tamanho na ordem de 50% (Rayside et al., 1999). Essa otimização poderá compensar o crescimento do arquivo devido ao aumento no tamanho das instruções, quando comparadas às de Java.

A tabela de constantes será agrupada por tipos (`int`, `Object`[5], `double`, `long`), resultando em uma melhor compactação (Pugh, 1999). A descrição dos métodos, com seu nome, tipos de parâmetros e retorno, também será desmembrada em referências para a tabela de constantes (Rayside et al., 1999).

Os tipos primitivos e os nomes das classes do pacote `java.lang` serão armazenados internamente na máquina virtual, eliminando assim a necessidade de repeti-los na tabela de constantes dos programas.

3.6. Arquitetura de palavra das instruções

A maioria dos processadores não consegue ler uma palavra em um endereço que esteja desalinhado em relação à palavra do processador. Por exemplo, a maioria dos processadores com palavras de 32 bits só consegue ler valores de 32 bits que estejam em endereços múltiplos de quatro. Caso não estejam alinhados, para ler um inteiro qualquer, é preciso lê-lo byte a byte e fazer um deslocamento de bits, remontando o valor original. O código gerado pelo compilador deverá garantir o alinhamento, de forma a permitir a leitura da instrução de uma só vez.

O tamanho das instruções será sempre de múltiplos de 32 bits, e o formato de armazenamento dos números seguirá a ordem *little-endian*. Desta forma, em arquiteturas little-endian, que correspondem à maioria das usadas atualmente em computadores de mesa ou móveis, como ARM e x86, não serão necessárias

---

[5] Apenas objetos do tipo `String` são armazenados na tabela de contantes.

leituras byte a byte das instruções como ocorre no Java, que optou pela arquitetura big-endian. Caso o programa seja executado em um processador de arquitetura diferente, a conversão será feita no carregamento da classe.

3.7. Formato das instruções

Conforme visto na seção anterior, o tamanho das instruções foi fixado em 32 bits. As únicas instruções que podem ocupar múltiplos de 32 bits são as de chamada de método e a de `switch`.

As instruções seguem o seguinte formato: os primeiros 8 bits contêm o índice das 256 operações possíveis, e os 24 bits restantes contêm as informações dos operandos. A seguir um resumo dos principais grupos de instruções definidos:

Operação	Formato dos operandos
Movimentação	destino, origem
Aritméticas e lógicas	destino, operando1, operando2
Desvio condicional	operando1, operando2, deslocamento
Conversão entre tipos	tipo_destino, tipo_origem
Retorno de método	operando1

3.8. Escolha das instruções

Definimos que poderá haver no máximo 256 instruções. Consequentemente, não há espaço para implementar todos os tipos de operandos para todas as operações disponíveis. O essencial é que seja possível mover todos os tipos de operandos para um registrador, e que as outras instruções operem através de registradores. Dessa forma, o incremento de um campo interno (*da instância*) tem a seguinte seqüência de instruções:

```
mov reg, campo_interno
add reg, reg, 1
mov campo_interno, reg
```

Porém, restringir que apenas a instrução de movimentação tenha acesso a campo e a vetor iria de encontro com as sugestões dos artigos descritos na seção anterior, diminuindo o desempenho da máquina virtual. Se houvesse uma instrução que permitisse a adição de um número a um campo, guardando o re-

sultado nesse campo, trocaríamos as três instruções no exemplo acima por apenas uma. Mas qual é a freqüência que esta operação ocorre nos programas Java? Faria sentido criar uma instrução de movimentação de um campo externo diretamente para outro campo externo? Ou somar um campo interno a um número, guardando o resultado em um vetor? Em virtude das dúvidas sobre quais operandos criar, além dos essenciais, e para quais operações, decidimos escrever um interpretador que possibilitasse a geração de estatísticas de forma a definir melhor o conjunto de instruções.

Esse interpretador efetua uma análise *estática* no código, convertendo os bytecodes Java para o formato de instruções que idealizamos. De forma a facilitar o trabalho, optamos por escrever o interpretador em Java, e usar a biblioteca Byte Code Engineering Library (BCEL), que, a partir de um código compilado Java, dispõe os métodos e bytecodes como objetos onde seus operandos podem ser facilmente consultados.

A seguir descreveremos o funcionamento do interpretador.

O interpretador analisa os bytecodes seguindo o fluxo do programa, passando pelos blocos *if* e *else*, além dos outros blocos de código. Desvios são seguidos e, caso um laço seja detectado, ele continua como se não houvesse a instrução do laço. Os seguintes tipos de blocos são ignorados, porque a inclusão deles aumentaria bastante a complexidade do interpretador:

- O termo `senão` no operador ternário "`condição ? então : senão`" - esse código é ignorado porque o operador ternário produz variáveis temporárias, e teria que haver um controle da pilha para executar o `então`, voltar o estado, e executar o `senão`.
- Blocos `finally`[6] – optamos por não implementar os opcodes jsr e ret, que são usados no `finally`.
- Alguns casos de `switch/case`, como quando o `default` é definido antes do último `case`; nesse caso, todos os `case`s subseqüentes são ignorados.

Devido a essas limitações, pouco menos de 10% do código deixou de ser analisado. Acreditamos, porém, que isso não invalida os resultados obtidos, porque o código que aparece nestes blocos não é muito diferente do que aparece no resto do programa.

Nas estatísticas são coletados os totais abaixo:

- Classes, métodos, bytecodes interpretados e não interpretados, e a estimativa do número de instruções (incluindo o tamanho em bytes ocupado por elas).

---

[6] O tratamento de exceção em Java possui a seguinte sintaxe:
```
try {...} catch {...} finally {...}
```

- Operações aritméticas: mov, add, sub, mul, div, shr, shl, ushr, or, xor, and, mod.
- Operações de desvio condicional: jeq, jne, jlt, jle, jgt, jge.
- Casos em que o desvio condicional pula para um endereço fora da faixa que nossas instruções suportam. Para contornar esses casos, basta implementar o desvio para uma instrução `JMP`, que pularia para o endereço exigido.
- Variáveis usadas nos métodos, incluídas aí a soma do número de variáveis locais com o número de parâmetros.

Além de coletar estatísticas sobre as instruções, o interpretador obterá os tipos mais usados de operandos, que podem ser:

- Constante: valor numérico entre -2048 e +2047
- Símbolo: valor numérico fora da faixa acima ou uma *string*.
- Registrador: as variáveis locais e temporárias usadas em Java foram mapeadas para os registradores na nova especificação.
- Vetor
- Comprimento de vetor (*array length*)
- Campo normal desta classe.
- Campo estático desta classe.
- Campo normal de outra classe.
- Campo estático de outra classe.

Os tipos foram mapeados de acordo com a especificação Java (`int` para `int`, `long` para `long`, etc), exceto o tipo `float`, que foi substituído pelo `double`, pois na nossa especificação o `float` foi eliminado; e o tipo `boolean`, que foi mapeado para o `int` (uma variável do tipo `boolean` é armazenada como um `int`, mas um vetor de `boolean` é armazenado como um vetor de `byte`).

Foram usados dois conjuntos de programas: o código fonte do J2SE 5.0, que inclui o compilador, máquina virtual e as bibliotecas, compreendendo cerca de 12.700 classes; e programas Java diversos juntamente com uma biblioteca de classes cujo alvo são PDAs, com aproximadamente 1.400 classes.

O resumo dos resultados é exibido na tabela 7, contendo o total de métodos, bytecodes contabilizados e ignorados, uma estimativa do número de instruções na nova especificação, e a quantidade de instruções para desvio condicional cujo endereço destino é maior que o suportado pelas nossas instruções (e que, portanto, requerem uma codificação extra).

As tabelas 8 e 9 mostram as operações aritméticas e lógicas que representam pelo menos 1% do total, primeiro para os programas Java, e em seguida, para o J2SE 5.0. Por elas vemos que as instruções mais usadas são de adição e subtração. Vale notar que a instrução

```
REG = REG SUB CONST
```

... equivale a

```
REG = REG ADD -CONST.
```

Logo, somando as ocorrências dessas duas instruções, temos que REG = REG ADD CONST atinge 22% na tabela 8 e 36% na tabela 9, o que para nós foi uma surpresa, pois imaginávamos que instruções somente com registradores ocorreriam mais.

Foi obtido um total de 371 instruções aritméticas e lógicas na análise dos programas Java. Dessas, 71% foram do tipo int, 14% do tipo long e 13% do tipo double. Para o J2SE 5.0, o total foi de 621 instruções, sendo 57% do tipo int, 25% do tipo long e 16% do double. Essa predominância do tipo int explica porque o Java tem tantos bytecodes para o tipo (41 dos 202, ou 20%, excluídos os de conversão entre tipos). Para os programas Java, apenas 43 de um total de 5.482 instruções usaram o tipo float.

Podemos também ordenar as tabelas pelos tipos de operandos, listando apenas as instruções que alcançaram 1% (tabelas 10 e 11). Nestas duas tabelas, vemos que a maioria dos operandos é do tipo REG = REG op CONST. Portanto, é importante garantir que todas as instruções da especificação possam trabalhar diretamente com constantes.

As tabelas 12 e 13 exibem os resultados obtidos para as instruções de desvio condicional. Nas instruções com o tipo Object, CONST se refere ao null de Java. Portanto, cerca de 15% (nos programas Java) e 22% (no J2SE 5.0) das operações de desvio condicional são para verificar se um objeto é ou não nulo (e desviar, caso positivo).

As tabelas 14 e 15 ordenam as instruções de desvio condicional por operando. Observando estas duas tabelas, vemos novamente a grande incidência de comparações entre registrador com constante e registrador com registrador, variando de 77% a 80%. É interessante notar também a presença da instrução REG JGE TAM_VET, que é usado na implementação de laços do tipo:

```
for (int i = 0; i < vetor.length; i++)
```

Por fim, listamos nas tabelas 16 e 17 a soma do número de variáveis temporárias com o número de parâmetros fornecidos para os métodos, que denominamos *variáveis locais*. Nosso objetivo é confirmar que os 64 registradores disponíveis para cada tipo são suficientes para atender aos programas testados.

Aproximadamente 97% dos métodos nos programas Java tiveram até 10 variáveis locais. Apenas 15 métodos, de um total de 12.411, tinham entre 36 e

60 locais. No caso extremo, havia 2 métodos com 94 locais: esses métodos recebiam 40 parâmetros, o que é muito difícil de ocorrer[7].

Em relação aos programas J2SE 5.0, 99.9% dos 115.495 métodos possuíam até 10 locais, e apenas um método atingiu a marca de 26 variáveis.

Estes resultados confirmam que o uso de até 64 registradores por cada um dos quatro tipos, atingindo um máximo de 256 registradores, atende à absoluta maioria (acima de 99,9%) dos programas testados.

Baseados nos resultados obtidos com o interpretador, decidimos implementar todas as instruções aritméticas, lógicas e de desvio condicional com mais de 1% de ocorrência em cada um dos conjuntos de programas testados. A única exceção ficará com a operação `REG = CONST MUL EST_INTR` (que está realçada na tabela 8), pois não há espaço na instrução para todos esses operandos. Todas as instruções poderão operar também com registradores e constantes.

Programas Usados	Classes	Métodos	Bytecodes contabilizados	Bytecodes ignorados	Instruções	Desvios excedidos
Programas Java e bibliotecas de classe cujo alvo são PDAs.	1.429	12.411	615.529 (1.245.621 bytes)	66.751 (10%)	398.827 (1.595.308 bytes)	15 em 27.362
J2SE 5.0, que inclui o compilador, máquina virtual e as bibliotecas.	12.781	115.495	3.025.957 (5.757.972 bytes)	105.964 (3%)	2.104.118 (8.416.472 bytes)	1 em 194.141

Tabela 7: Programas usados durante o teste e resumo dos resultados

[7] Os métodos foram implementados assim porque seus programadores não conheciam direito a plataforma que estavam utilizando. Portanto, se a linguagem os obrigasse a utilizar menos parâmetros, é possível que eles tivessem pesquisado um pouco mais e programado corretamente, resolvendo a questão sem usar parâmetro algum.

Inst.	Tipo	Operandos	Qtde.	%
ADD	Int	REG = REG op CONST	2.372	16,06
ADD	Int	REG = REG op REG	2.079	14,07
SUB	Int	REG = REG op REG	1.023	6,92
SUB	Int	REG = REG op CONST	925	6,26
MUL	Int	REG = REG op CONST	758	5,13
ADD	Int	C_INTR = REG op CONST	461	3,12
DIV	Int	REG = REG op CONST	397	2,68
AND	Int	REG = REG op CONST	348	2,35
SUB	Int	REG = VETOR op CONST	285	1,92
SHL	Int	REG = REG op CONST	274	1,85
SHR	Int	REG = REG op CONST	219	1,48
MUL	*Int*	*REG = CONST op EST_INTR*	*206*	*1,39*
SUB	Int	REG = CONST op REG	187	1,26
ADD	Int	REG = REG op SIMB	165	1,11
ADD	Int	C_INTR = REG op REG	150	1,01
ADD	Int	VETOR = REG op CONST	150	1,01

Tabela 8: Instruções aritméticas e lógicas dos programas Java
que alcançaram 1% do total (10.033 instruções, ou 67,84% do total contabilizado).

Inst.	Tipo	Operandos	Qtde	%
ADD	Int	REG = REG op CONST	18.262	27,11
ADD	Int	REG = REG op REG	7.445	11,05
SUB	Int	REG = REG op CONST	6.048	8,97
SUB	Int	REG = REG op REG	4.533	6,73
AND	Int	REG = REG op CONST	2.549	3,78
ADD	Int	C_INTR = REG op CONST	1.602	2,37
ADD	Int	VETOR = REG op CONST	1.440	2,13
SHL	Int	REG = REG op CONST	1.204	1,78
MUL	Int	REG = REG op REG	920	1,36
SHR	Int	REG = REG op CONST	853	1,26
AND	Int	REG = VETOR op CONST	790	1,17
MUL	Int	REG = REG op CONST	784	1,16
SUB	Int	REG = CONST op REG	783	1,16
OR	Int	REG = REG op REG	729	1,08
ADD	Dbl	REG = REG op REG	711	1,05
MUL	Dbl	REG = REG op REG	692	1,02

Tabela 9: Instruções aritméticas e lógicas do J2SE 5.0
que alcançaram 1% do total (49.345 instruções, ou 73,18% do total contabilizado).

Operandos	Qtde	%	Instruções
REG = REG op CONST	5.293	35,83	ADD AND DIV MUL SHL SHR SUB
REG = REG op REG	3.102	21,00	ADD SUB
C_INTR = REG op CONST	461	3,12	ADD
REG = VETOR op CONST	285	1,92	SUB
REG = CONST op EST_INTR	206	1,39	MUL
REG = CONST op REG	187	1,26	SUB
REG = REG op SIMB	165	1,11	ADD
C_INTR = REG op REG	150	1,01	ADD
VETOR = REG op CONST	150	1,01	ADD

Tabela 10: Operandos das instruções aritméticas e lógicas para Java que tenham atingido 1% do total. Todas as instruções são do tipo `int`.

Operandos	Qtde	%	Instruções
REG = REG op CONST	29.700	44,09	ADD AND MUL SHL SHR SUB
REG = REG op REG	15.030	22,31	*ADD MUL* ADD MUL OR SUB
C_INTR = REG op CONST	1.602	2,37	ADD
VETOR = REG op CONST	1.440	2,13	ADD
REG = CONST op REG	790	1,17	AND
REG = VETOR op CONST	783	1,16	SUB

Tabela 11: Operandos das instruções aritméticas e lógicas para J2SE 5.0 que tenham atingido 1% do total. Todas as instruções são do tipo `int`, exceto as em itálico, que são do tipo `double`.

Inst.	Tipo	Operandos	Qtde	%
JEQ	Obj	REG op CONST	1.836	9,69
JNE	Int	REG op CONST	1.670	8,82
JEQ	Int	REG op REG	1.195	6,31
JNE	Obj	REG op REG	1.043	5,51
JNE	Int	REG op REG	994	5,25
JNE	Obj	REG op CONST	952	5,02
JEQ	Int	REG op SIMB	929	4,90
JLT	Int	REG op REG	920	4,86
JEQ	Int	REG op CONST	910	4,80
JNE	Int	REG op SIMB	826	4,36
JEQ	Int	REG op C_INTR	811	4,28
JGE	Int	REG op REG	619	3,27
JLE	Int	REG op REG	613	3,23
JNE	Int	REG op C_INTR	488	2,57
JLE	Int	REG op CONST	464	2,45
JGT	Int	REG op REG	358	1,89
JLT	Int	REG op CONST	335	1,76
JEQ	Int	REG op C_EXTN	255	1,34
JNE	Int	REG op C_EXTN	191	1,00

Tabela 12: Instruções de desvio condicional para Java
que tenham atingido 1% do total (15.409 instruções, ou 81,31% do total contabilizado).

Inst.	Tipo	Operandos	Qtde	%
JEQ	Obj	REG op CONST	19.244	13,85
JEQ	Int	REG op REG	13.418	9,66
JNE	Obj	REG op CONST	11.919	8,58
JNE	Int	REG op REG	8.413	6,05
JGE	Int	REG op REG	8.345	6,01
JNE	Int	REG op CONST	8.049	5,79
JEQ	Int	REG op CONST	6.692	4,81
JEQ	Int	REG op C_INTR	6.292	4,53
JEQ	Int	REG op SIMB	4.386	3,15
JEQ	Obj	REG op REG	4.146	2,98
JLE	Int	REG op REG	3.132	2,25
JGE	Int	REG op TAM_VET	3.131	2,25
JNE	Int	REG op C_INTR	2.980	2,14
JLT	Int	REG op REG	2.489	1,79
JNE	Obj	REG op REG	2.450	1,76
JNE	Int	REG op SIMB	1.967	1,41
JLE	Int	REG op CONST	1.532	1,10
JLT	Int	REG op CONST	1.516	1,09
JEQ	Int	REG op EST_EXT	1.401	1,00

Tabela 13: Instruções de desvio condicional para J2SE 5.0
que tenham atingido 1% do total (111.502 instruções, ou 80,2% do total contabilizado).

Operandos	Qtde	%	Instruções
REG op CONST	6.167	32,58	*JEQ JNE* JEQ JNE JLE JLT
REG op REG	5.742	30,33	JEQ JNE JGE JLE JGT JLT *JNE*
REG op SIMB	1.755	9,27	JEQ JNE
REG op C_INTR	1.299	6,86	JEQ JNE
REG op EST_EXT	446	2,35	JEQ JNE

Tabela 14: Operandos das instruções de desvio condicional para Java
que tenham atingido 1% do total. Todas as instruções são do tipo `int`,
com exceção das em itálico, que são do tipo `Object`.

Operandos	Qtde	%	Instruções
REG op CONST	48.952	35,25	*JEQ JNE* JEQ JNE JLE JLT
REG op REG	42.393	30,53	*JEQ JNE* JEQ JNE JGE JLE JLT
REG op C_INTR	9.272	6,67	JEQ JNE
REG op SIMB	6.353	4,57	JEQ JNE
REG op TAM_VETOR	3.131	2,25	JGE
REG op EST_EXT	1.401	1,00	JEQ

Tabela 15: Operandos das instruções de desvio condicional para J2SE 5.0 que tenham atingido 1% do total. Todas as instruções são do tipo `int`, com exceção das em itálico, que são do tipo `Object`.

Nr. Locais	0	1	2	3	4	5	6	7	8	9	10	>10
% Métodos	31,0	31,1	9,9	10,9	3,2	4,0	1,7	2,4	1,3	1,2	0,5	2,8

Tabela 16: Número de locais por método para programas Java

Nr. Locais	0	1	2	3	4	5	6	7	8	9	10	>10
% Métodos	38,5	36,5	13,9	5,8	2,4	1,1	1,0	0,3	0,2	0,1	0,1	0,1

Tabela 17: Número de locais por método para J2SE 5.0

3.9. Listagem das instruções

Listaremos a seguir as instruções que definimos a partir das estatísticas coletadas.

Na tabela 18 é exibida a lista dos operandos usados nas instruções, o seu significado, o número de bits requeridos e os parâmetros necessários. Cabe lembrar que todas as instruções têm 32 bits, com exceção das de chamada de método (`callInternal` e `callExternal`) e a de `switch`, que podem ter mais de um bloco de 32 bits (as de chamada para indicar os parâmetros, e a de switch para indicar os valores e endereços das opções).

Para operações em vetores, duas categorias de instruções foram definidas: uma que efetua testes de limite e de ponteiro nulo, e outra que não efetua

estes testes. Dessa forma, o compilador especifica qual o tipo de acesso a vetor que pode ser feito; em muitos casos é trivial a verificação que o índice nunca extrapola o limite, e nestes casos o desempenho melhorará. A nulidade do vetor deverá ser verificada antes do início do laço através da instrução `TEST_regO`. Vetores de byte, short e char possuem instruções específicas para movimentação de valores, de forma que não seja necessária a checagem do tipo do vetor.

A tabela 19 exibe as instruções para a operação `MOV`, que move o operando da direita para o operando à esquerda. O tipo de registrador origem ou destino define o tipo de dado usado na operação. Não é possível mover de um tipo para outro, como, por exemplo, de `int` para `double`, usando a operação `MOV`, para isso existem instruções específicas (`CONV`).

A tabela 20 exibe as instruções para operações aritméticas e lógicas. Essas instruções possuem o formato `Destino = origem1 op origem2`. O tipo inteiro possui algumas instruções aritméticas para acesso a campo e a vetor, conforme sugerido nos artigos. Optamos por não implementar esse acesso a outros tipos porque a ocorrência dessas instruções foi muito baixa. A instrução `REG = VETOR SUB CONST`, que consta na tabela 8 com 1,92% de ocorrências, foi substituída pela análoga `REG = VETOR ADD CONST`.

A tabela 21 exibe as instruções para as operações de desvio condicional, que desvia para um endereço relativo ao atual quando a condição for satisfeita. . O deslocamento informado é em múltiplos de 32 bits, sendo que a maioria das instruções usa 12 bits, o que dá uma faixa de -2048 a +2047. Uma única instrução possui um deslocamento de 6 bits, o que dá uma faixa de -32 a +31.

A tabela 22 exibe as instruções para chamada de método. Duas instruções foram criadas: `CALL_INTERNAL` e `CALL_EXTERNAL`. A primeira é usada para chamar métodos da classe que está sendo executada ou de alguma classe herdada, e a segunda, para chamar um método de outra classe. A instrução possui os seguintes operandos:

Operando	**Uso**
símbolo	Se for um método interno, contém o índice de 12 bits para a tabela de métodos da classe. Se for um método externo, é fornecido o índice na tabela de constantes onde está armazenada a descrição completa do método, incluindo o nome da classe, do método e seus parâmetros.
this	Se o método não for estático, contém o registrador de objeto que armazena a instância.
retorno ou parâmetro	Se o método retornar algo, informa o registrador onde se deseja que a resposta seja copiada. Se nada for retornado e o método tiver parâmetros, conterá o primeiro parâmetro.

Os demais parâmetros do método, se houverem, serão passados nas palavras de 32 bits seguintes. Cada parâmetro ocupa 8 bits, resultando em um valor na faixa de 0 a 255, codificado de acordo com a tabela abaixo:

0 <= valor <= 63	Índice do registrador onde se encontra o parâmetro
64 <= valor <= 255	Número inteiro, calculado de acordo com a fórmula: `número = valor - 65` O número pode variar de -1 a 190, e pode ser armazenado em registradores dos tipos int, double e long. Se o parâmetro for do tipo `Object`, então assume-se que `null` deverá ser retornado ao invés de um número.

Com a codificação acima, evitamos mover para um registrador as constantes null, -1, 0, 1... quando forem passadas como parâmetro na chamada do método. Note que essa codificação não é válida para o primeiro parâmetro, pois este é armazenado na instrução (retorno ou parâmetro), e portanto tem apenas 6 bits disponíveis (o suficiente apenas para indicar o registrador onde se encontra o parâmetro). O tipo de cada parâmetro é obtido a partir da estrutura que armazena as informações do método.

A tabela 23 mostra as instruções para conversão entre tipos de dados. Esses são os únicos que permitem a passagem de valores entre os diferentes tipos de registradores.

A tabela 24 mostra as formas disponíveis para retorno de método, e a 25 exibe outras instruções que não se encaixam nos grupos anteriores.

Por fim, a tabela 33 do apêndice mostra o mapeamento dos bytecodes Java para as instruções aqui definidas.

No próximo capítulo, iremos mostrar testes que avaliam essa especificação.

Operando	Significado	Nr. Bits	Parâmetros
regI	Registrador para inteiro	6	Número do registrador de 0 a 63
regO	Registrador para Object	6	
regD	Registrador para double	6	
regL	Registrador para long	6	
cpIntr	Acesso a campo interno, não estático, da classe atual (variável de instância).	16	Índice na tabela de constantes para a definição do campo.
estIntr	Acesso a campo estático da classe atual (variável de classe).	16	
cpExtr	Acesso a campo não estático de outra classe (variável de instância).	18	Índice na tabela de constantes para a definição do campo. A tabela de constantes é obtida a partir do ponteiro para o objeto passado em regO.
estExtr	Acesso a campo estático de outra classe (variável de classe).	18	
vets	Acesso a vetor *sem* verificação de ponteiro nulo ou limite estourado.	12	Índice na tabela de constantes para a definição do vetor. Número do registrador inteiro (regI) para o índice no vetor.
vetc	Acesso a vetor *com* verificação de ponteiro nulo e limite estourado.	12	
regIb	Registrador para inteiro onde o conteúdo é um byte 8 bits com sinal	6	Número do registrador
regIs	Registrador para inteiro onde o conteúdo é um short 16 bits com sinal	6	
regIc	Registrador para inteiro onde o conteúdo é um char 16 bits sem sinal	6	
simb	Acesso à tabela de constantes	12	Índice na tabela de constantes para a definição do item.
s12	Número inteiro de 12 bits com sinal.	12	O número propriamente dito.
s8	Número inteiro de 8 bits com sinal.	8	
s8 B	Número de 8 bits com sinal para ser usado em vetores de byte	8	
s12 CSIDL	Número de 12 bits com sinal para ser usado em vetores de char/short/int/ double/long	12	
delta	Deslocamento relativo (normal) usado nas operações de desvio condicional.	12	O número de linhas (múltiplos de 32 bits) a serem desviadas a partir da linha atual. Um bit é usado para especificar a direção.
sdelta	Deslocamento relativo (curto) usado nas operações de desvio condicional.	6	
tamvet	Tamanho do vetor	6	ïndice do registrador de objeto (regO) para a referência do vetor.

Tabela 18: Operandos usados nas instruções

#	Operação	Tipo
1	regI = regI	int
2	regI = cpIntr	int
3	regI = cpExtr	int
4	regI = estIntr	int
5	regI = estExtr	int
6	regI = vets	int
7	regI = vetc	int
8	regI = simb	int
9	regI = s18	int
10	regI = tamvet	int
11	regO = regO	Object
12	regO = cpIntr	Object
13	regO = cpExtr	Object
14	regO = estIntr	Object
15	regO =estExtr	Object
16	regO = vets	Object
17	regO = vetc	Object
18	regO = simb	Object
19	regD = regD	double
20	regD = cpIntr	double
21	regD = cpExtr	double
22	regD = estIntr	double
23	regD = estExtr	double
24	regD = vets	double

#	Operação	Tipo
25	regD = vetc	double
26	regD = simb	double
27	regD = s18	double
28	regL = regL	long
29	regL = cpIntr	long
30	regL = cpExtr	long
31	regL = estIntr	long
32	regL = estExtr	long
33	regL = vets	long
34	regL = vetc	long
35	regL = simb	long
36	regL = s18	long
37	cpIntr = regI	int
38	cpIntr = regO	Object
39	cpIntr = regD	double
40	cpIntr = regL	long
41	cpExtr = regI	int
42	cpExtr = regO	Object
43	cpExtr = regD	double
44	cpExtr = regL	long
45	estIntr = regI	int
46	estIntr = regO	Object
47	estIntr = regD	double
48	estIntr = regL	long

#	Operação	Tipo
49	estExtr = regI	int
50	estExtr =regO	Object
51	estExtr =regD	double
52	estExtr = regL	long
53	vetc = regI	int
54	vetc = regO	Object
55	vetc = regD	double
56	vetc = regL	long
57	vets = regI	int
58	vets = regO	Object
59	vets = regD	double
60	vets = regL	long
61	vetc = regIb	byte
62	vetc = regIs	short
63	vetc = regIc	char
64	vets = regIb	byte
65	vets = regIc	char
66	regIb = vetc	byte
67	regis = vetc	short
68	regIc = vetc	char
69	regIb = vets	byte
70	regIc = vets	char
175	regO = null	Object

Tabela 19: Instruções de movimentação

#	Instr	Operação	Tipo
71	ADD	regI = regI + regI	int
72	ADD	regI = s12 + regI	int
73	ADD	regI = vetc + s6	int
74	ADD	regI = vets + s6	int
75	ADD	regI = regI + simb	int
76	ADD	regD = regD + regD	double
77	ADD	regL = regL + regL	long
78	ADD	vets = regI + s6	int
79	ADD	cpIntr = regI + regI	int
80	ADD	cpIntr = regI + s6	int
81	SUB	regI = s12 - regI	int
82	SUB	regI = regI - regI	int
83	SUB	regD = regD - regD	double
84	SUB	regL = regL - regL	long
85	MUL	regI = regI * s12	int
86	MUL	regI = regI * regI	int
87	MUL	regD = regD * regD	double
88	MUL	regL = regL * regL	long
89	DIV	regI = regI / s12	int
90	DIV	regI = regI / regI	int
91	DIV	regD = regD / regD	double
92	DIV	regL = regL / regL	long
93	MOD	regI = regI % s12	int
94	MOD	regI = regI % regI	int
95	MOD	regD = regD % regD	double
96	MOD	regL = regL % regL	long
97	SHR	regI = regI >> s12	int
98	SHR	regI = regI >> regI	int
99	SHR	regL = regL >> regL	long
100	SHL	regI = regI << s12	int
101	SHL	regI = regI << regI	int
102	SHL	regL = regL << regL	long
103	USHR	regI = regI >>> s12	int
104	USHR	regI = regI >>> regI	int
105	USHR	regL = regL >>> regL	long
106	AND	regI = regI & s12	int
107	AND	regI = vets & s6	int
108	AND	regI = regI & regI	int
109	AND	regL = regL & regL	long
110	OR	regI = regI | s12	int
111	OR	regI = regI | regI	int
112	OR	regL = regL | regL	long
113	XOR	regI = regI ^ s12	int
114	XOR	regI = regI ^ regI	int
115	XOR	regL = regL ^ regL	long
186	INC	regI, s16	int

Tabela 20: Instruções aritméticas e lógicas

#	Instr	Operação	Tipo
116	JEQ	if (regO == regO) ip+=delta	Object
117	JEQ	if (regO == null) ip+=delta	Object
118	JEQ	if (regI == regI) ip+=delta	int
119	JEQ	if (regL == regL) ip+=delta	long
120	JEQ	if (regD == regD) ip+=delta	double
121	JEQ	if (regI == s6) ip+=delta	int
122	JEQ	if (regI == simb) ip+=sdelta	int
123	JEQ	if (regI == cpIntr) ip+=delta	int
124	JNE	if (regO != regO) ip+=delta	Object
125	JNE	if (regO != null) ip+=delta	Object
126	JNE	if (regI != regI) ip+=delta	int
127	JNE	if (regL != regL) ip+=delta	long
128	JNE	if (regD != regD) ip+=delta	double
129	JNE	if (regI != s6) ip+=delta	int
130	JNE	if (regI != simb) ip+=delta	int
131	JNE	if (regI != cpIntr) ip+=delta	int
132	JLT	if (regI < regI) ip+=delta	int
133	JLT	if (regL < regL) ip+=delta	long
134	JLT	if (regD < regD) ip+=delta	double
135	JLT	if (regI < s6) ip+=delta	int
136	JLE	if (regI <= regI) ip+=delta	int
137	JLE	if (regL <= regL) ip+=delta	long
138	JLE	if (regD <= regD) ip+=delta	double
139	JLE	if (regI <= s6) ip+=delta	int
140	JLE	if (regI <= cpIntr) ip+=delta	int
141	JGT	if (regI > regI) ip+=delta	int
142	JGT	if (regL > regL) ip+=delta	long
143	JGT	if (regD > regD) ip+=delta	double
144	JGT	if (regI > s6) ip+=delta	int
145	JGE	if (regI >= regI) ip+=delta	int
146	JGE	if (regL >= regL) ip+=delta	long
147	JGE	if (regD >= regD) ip+=delta	double
148	JGE	if (regI >= s6) ip+=delta	int
149	JGE	if (regI >= tamvet) ip+=delta	int
187	DECJGTZ	if (--regI > 0) ip += delta	int
188	DECJGEZ	if (--regI >= 0) ip += delta	int

Tabela 21: Instruções de desvio condicional
O deslocamento `delta` tem 12 bits com sinal, e `sdelta` tem 6 bits com sinal.

#	Instrução	Parâmetros
173	CallInternal	[this], [param1...]
174	CallExternal	[this], [param1...]

Tabela 22: Instruções de chamada de método

#	Operação	Tipos envolvidos
150	regL = (long)regI	long, int
151	regD = (double)regI	double, int
152	regI = (byte)regI	int, byte
153	regI = (char)regI	int, char
154	regI = (short)regI	int, short
155	regI = (int)regL	int, long
156	regD = (double)regL	double, long
157	regI = (int)regD	int, double
158	regL = (long)regD	long, double

Tabela 23: Instruções de conversão entre tipos de dados

#	Operação	Tipos envolvidos
177	return void	nenhum
159	return regI	byte, short, char, int
160	return regO	Object
161	return regD	double
162	return regL	long
178	return s24I	byte, short, char, int
179	return null	Object
180	return s24D	double
181	return s24L	long
182	return simb16I	byte, short, char, int
183	return simb16O	Object
184	return simb16D	double
185	return simb16L	long

Tabela 24: Instruções para retorno de método

#	Operação	Descrição
163	Jump s24	Desvio incondicional para um endereço relativo de 23 bits com sinal (-8.388.608 a 8.388.607)
164	Switch (regI)	Implementação do switch/case. O registrador inteiro contém o valor do switch. A lista de valores possíveis mais os endereços para desvio relativo são armazenados na tabela de constantes, cujo índice é passado como parâmetro.
165	SyncStart regO	Inicio de sincronismo, baseado no objeto passado como parâmetro.
166	SyncEnd regO	Fim de sincronismo, baseado no objeto passado como parâmetro.
167	NewArray tipo, u16	Criação de array do tipo especificado (0=int, 1=object, 2=double, 3=long, 4=byte, 5=short, 6=char), cujo tamanho é passado como constante numérica de 16 bits sem sinal.
168	NewArray tipo, regI	Criação de vetor do tipo especificado, cujo tamanho é passado no registrador inteiro
169	NewObj simb, regO	Criação de objeto, cujo tipo é especificado pelo índice na tabela de constantes fornecido, e armazenando o resultado no registrador de objeto passado como parâmetro.
170	Throw simb	Disparo de exceção, cujo tipo é especificado pelo índice na tabela de constantes fornecido.
171	Instanceof regO, simb, regI	Verifica se o objeto fornecido é compatível com o tipo especificado na tabela de constantes, armazenando o resultado no registrador inteiro fornecido.
172	Checkcast regO, simb	Verifica se o objeto fornecido é compatível com o tipo especificado na tabela de constantes. Se não for, uma exceção *ClassCastException* é disparada.
176	Test regO	Verifica se o objeto fornecido é nulo, disparando uma exceção se o for.
0	break	Interrompe a execução da máquina virtual e desvia para um depurador.

Tabela 25: Instruções não categorizadas

4
Avaliação do Código Gerado

Nós fizemos alguns exemplos para avaliar a eficiência da especificação proposta, tanto em termos de velocidade de execução quanto de diminuição do tamanho do código. Criamos um compilador para ler um arquivo Java e gerar código de acordo com a especificação. Além dele, também criamos uma máquina virtual que pudesse executar o código gerado pelo compilador. Ambos são um pouco rudimentares, mas ajudarão a verificar se a especificação é viável. Nós a denominamos VERA (*Very Enhanced Runtime Architecture*) [8].

4.1. Descrição dos testes

Os testes foram realizados em dois computadores: o primeiro, um *notebook* Toshiba 5205-s119, com processador Pentium 4-M de 2.2 GHz, 1 GB de RAM, rodando Windows 2000, com a rede desconectada e apenas com os serviços essenciais em execução, o que garante que outras tarefas não interfiram nos testes; o segundo, um PDA Dell Axim X3 com 300 MHz e 64MB de RAM, rodando Windows CE 4.2.

As máquinas virtuais e os compiladores utilizados são exibidos na tabela abaixo:

Compilador	**Parâmetros**	**Máquina Virtual**	**Parâmetros**
JavaC do JDK 1.2.2	`-g:none`	Java VM do JDK 1.2.2	`-Djava.compiler`
JavaC do JDK 1.6.0	`-g:none`	Java VM do JDK 1.6.0	`-Djava.compiler`
JavaC do JDK 1.2.2	`-g:none`	IBM J9 6.1 CDC 1.0	nenhum
JavaC do JDK 1.2.2	`-g:none` `-target 1.1`	SuperWaba VM 5.71	nenhum
VERA	nenhum	VERA VM	nenhum

O JDK é a implementação Java oficial da Sun. Escolhemos duas versões dele porque sabemos que existem diferenças de desempenho, nem sempre fa-

voráveis à versão 1.6.0. O SuperWaba VM é uma máquina virtual criada para ser executada em PDAs, e que também pode ser executada no Windows. O IBM J9® é uma implementação de Java para Windows Mobile com a configuração *Connected Device Configuration* (CDC) versão 1.0.

Nosso objetivo foi verificar o desempenho de interpretadores, pois a máquina virtual VERA implementada não possui compilação sob demanda (JIT). Vale lembrar que a especificação foi criada justamente para melhorar o desempenho sem a necessidade de se escrever um JIT. Sabemos que o JDK possui um, e usamos a opção `-Djava.compiler` para desativa-lo.

Na compilação dos arquivos de teste com o JavaC do JDK, fornecemos ao compilador um parâmetro para não inserir símbolos de depuração (`-g:none`), diminuindo assim o tamanho do arquivo. Por sua vez, o SuperWaba VM requer que o arquivo `class` gerado seja compatível com a versão 1.1, e portanto passamos o parâmetro `-target 1.1`. Cabe ressaltar que tanto o compilador do JDK quanto do VERA não efetuam otimizações [9].

O código dos testes não varia entre as plataformas; o que varia é a declaração da classe, como obter o tempo atual em milisegundos (método `getTimeStamp`), e como exibir o resultado (método `print`). Portanto, foram criados três programas, um para cada plataforma (JDK, SuperWaba e VERA). As partes que diferem em cada plataforma são exibidas na tabela 26. A tabela 27 exibe as partes comuns, como a chamada aos testes pelo construtor, além de campos e métodos usados por alguns testes. A tabela 28 exibe o código Java dos testes propriamente ditos. A tabela 34 do apêndice exibe o código gerado pelo compilador VERA para o programa, ao lado dos bytecodes gerados pelo JavaC.

Efetuamos testes para manipulação de variáveis locais e de instância, chamadas a métodos passando parâmetros por valor e por variáveis locais, operações aritméticas em vetores, e chamadas a métodos *get* e *set*, muito comuns em programas orientados a objeto. Não foi possível fazer outros tipos de testes porque o compilador VERA ainda não suporta herança e tratamento de exceções[10], o que impossibilita que compilemos a biblioteca básica do Java e dessa forma utilizemos programas mais complexos.

---

[8] Na verdade, uma homenagem póstuma à mãe do autor.

[9] O JavaC possui uma opção -O, que foi usada apenas no compilador do JDK 1.1 para gerar código *inline*. A partir do JDK 1.2, todas as otimizações foram eliminadas, sendo deixadas a cargo do JIT, mas o parâmetro foi mantido por questões de compatibilidade. O uso do parâmetro não produz alterações nem mesmo no tamanho do arquivo.

[10] Esperamos uma piora praticamente desprezível no desempenho da máquina virtual quando implementarmos herança e tratamento de exceções.

Máquina Virtual	Listagem do Programa

JDK	public class AllTests {
	public static int getTimeStamp() {
	return (int)System.currentTimeMillis();
	}
	public static void print(int i) {
	System.out.println(i);
	}
	public static void main(String[] args) {
	new AllTests();
	}
	}

SuperWaba	public class AllTests extends waba.ui.MainWindow {
	public static int getTimeStamp() {
	return waba.sys.Vm.getTimeStamp();
	}
	public static void print(int i) {
	waba.sys.Vm.debug(""+i);
	}
	}

VERA	public class AllTests {
	public static native int getTimeStamp();
	public static native void print(int i);
	}

Tabela 26: Partes que diferem entre as plataformas

Descrição	Listagem do Programa

Construtor usado para chamar os testes e cronometrar o tempo de execução de cada um.	public AllTests()
	{
	int ini,fim;
	ini=getTimeStamp(); testLocal(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testFieldI(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testFieldF(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testFieldD(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testFieldL(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testMethod1(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testMethod2(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testArray(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testSet(); fim=getTimeStamp(); print(fim-ini);
	ini=getTimeStamp(); testGet(); fim=getTimeStamp(); print(fim-ini);
	}

Campos usados nos testes testFieldI, testFieldF, testFieldD e testFieldL.	private int fieldI; private float fieldF; private double fieldD; private long fieldL;

Método usado por testMethod1 e tesMmethod2.	public void method(int i, double d, long l, boolean b)
	{
	}

Campo e métodos usados em testSet e testGet.	private int age; public void setAge(int a) { age = a; } public int getAge() { return age; }

Tabela 27: Partes comuns às plataformas

Nome	**Descrição do Teste**	**Código Java**

TestLocal	Variáveis locais usando `for`.	`public void testLocal()`
		`{`
		`int j = 0;`
		`for (int i=10000000; i > 0; i--)`
		`j += 2;`
		`}`

TestFieldI	Acesso a campo do tipo `int`.	`public void testFieldI()`
		`{`
		`int c = 2;`
		`for (int i=10000000; i > 0; i--)`
		`fieldI += c;`
		`}`

TestFieldF	Acesso a campo do tipo `float`.
	Em VERA, esse tipo é convertido para `double`.
		`public void testFieldF()`
		`{`
		`float c = 2;`
		`for (int i=10000000; i > 0; i--)`
		`fieldF += c;`
		`}`

TestFieldD	Acesso a campo do tipo `double`.	`public void testFieldD()`
		`{`
		`double c = 2;`
		`for (int i=10000000; i > 0; i--)`
		`fieldD += c;`
		`}`

TestFieldL	Acesso a campo do tipo `long`.	`public void testFieldL()`
		`{`
		`long c = 2;`
		`for (int i=10000000; i > 0; i--)`
		`fieldL += c;`
		`}`

TestMethod1	Chamada a método usando números pequenos. Em VERA, nenhum registrador temporário é criado.	`public void testMethod1() {`
		`for (int i=1000000; i > 0; i--)`
		`method(50,30,150,false);`
		`}`

TestMethod2	Chamada a método usando locais.	`public void testMethod2() {`
		`int i1 = 500;`
		`double d1 = 300.123;`
		`long l1 = 1500L;`
		`boolean b1 = true;`
		`for (int i=1000000; i > 0; i--)`
		`method(i1,d1,l1,b1);`
		`}`

TestArray	Teste de array	`public void testArray() {`
		`int[] a = new int[10];`
		`for (int i=10000000; i > 0; i--)`
		`a[5] += i;`
		`}`

TestSet	Chamada a método *set*.	`public void testSet() {`
		`for (int i=1000000; i > 0; i--)`
		`setAge(10);`
		`}`

TestGet	Chamada a método *get*.	`public void testGet() {`
		`int a;`
		`for (int i=1000000; i > 0; i--)`
		`a = getAge();`
		`}`

Tabela 28: Código fonte dos testes realizados no Pentium 4-M.
No Dell Axim, os laços que iniciam em 10^7 nesta tabela foram reduzidos para 10^6.

4.2. Desempenho obtido nas plataformas

Na tabela 29, exibimos os resultados dos testes nas diversas plataformas rodando no Pentium 4-M. Os valores exibidos equivalem à média de 5 execuções do conjunto de testes. A figura 5 mostra um gráfico com o tempo total dos testes (em menor escala), junto com os resultados obtidos para cada teste.

Pela figura, podemos ver que o tempo de execução da máquina virtual VERA é um pouco mais que o dobro do tempo dos JDKs. Analisamos o código que a Sun disponibiliza[11], mas não conseguimos descobrir o porquê dessa diferença. Tanto a VERA VM quanto o código fonte disponibilizado usam *thread-dispatching* para escolher a próxima instrução. A VERA VM é escrita em C, e o Java VM em C++, que sabemos gerar um código mais lento. Isso nos induz a acreditar que alguma otimização pode estar sendo aplicada durante a interpretação do código. A suspeita aumenta quando verificamos os testes de chamada de método: Set, Get, Method1 e Method2. No caso do Set, o JDK 1.2.2 e o 1.6.0 praticamente se igualam. No Method1 e Method2, o 1.6.0 é 1,6 vezes mais rápido que o 1.2.2. Porém, no teste do Get, o tempo cai bruscamente: o 1.6.0 é 2,5 vezes mais rápido que o 1.2.2. É muito provável que alguma otimização tenha sido feita pela Java VM do 1.6.0 durante a interpretação do código para que isso ocorresse. Ficou ainda a dúvida se o código fonte disponibilizado é realmente o da máquina virtual que testamos.

Por outro lado, conhecemos bem o código fonte da SuperWaba VM. As diferenças principais entre a SuperWaba VM e a VERA VM são o uso de pilhas contra registradores, respectivamente, e o fato de a VERA VM usar *thread-dispatching*, ao contrário da SuperWaba VM para Windows, que usa um `switch`. Alterando a VERA VM para uso de `switch`, temos que o tempo total sobe para cerca de 14s (contra os 10s anteriores), ainda abaixo dos 19s obtidos pela SuperWaba VM. Com isso, podemos inferir que o uso de registradores proporciona um ganho de pelo menos 27% perante o uso de pilha.

Analisando novamente a figura 5, vemos que apenas no teste de Local a VERA VM teve um desempenho melhor que o JDK. A tabela 30 ajuda a entender por que: ela mostra o número de instruções que são repetidas em laço para cada

---

[11] http://download.java.net/jdk6. Esse código foi disponibilizado no final de 2006, portanto tivemos pouco tempo para analisá-lo.

teste que, em VERA, compreende as instruções entre `jump` e `decjgtz`, e no JDK, entre `goto` e `iinc+iload+ifgt`.

O teste de Local possui apenas uma instrução no laço. Os testes de campo e vetor requerem mais instruções para operarem, e isso aumenta proporcionalmente o tempo gasto para sua execução. Poderíamos facilmente melhorar os tempos da máquina virtual VERA na figura 5 adicionando instruções de incremento de campo para cada tipo de dado, diminuindo o número de instruções de três para apenas uma. Porém as análises do quão freqüentes essas instruções ocorrem, discutidas na seção 3.8, indicaram não valer a pena.

A instrução de chamada de método é bem mais complexa, pois requer a preparação do *frame* e a inserção dos parâmetros nos registradores, o que eleva em muito sua complexidade quando comparada com as outras instruções, e especialmente, quando comparada à maquina de pilha que não requer essa preparação.

Uma segunda seqüência de testes foi realizada em um PDA Dell Axim X3. Como a Sun não disponibiliza máquinas virtuais para estes equipamentos, utilizamos nos testes a máquina virtual da IBM, que é recomendada por muitos fabricantes de PDAs. Nessa plataforma, os resultados foram bem mais favoráveis à VERA, como se pode ver no gráfico da figura 6. Dessa figura tiramos duas importantes conclusões: em primeiro lugar, que o custo do JIT para converter o código para *assembler* parece não compensar[12], ocasionando uma piora no tempo de execução dos testes[13]; em segundo lugar, que a máquina de registradores gerou um ganho no desempenho quando comparado às de pilha.

Analisando os tempos na tabela 31, vemos que VERA só perde nas chamadas de método para a J9, e no teste do tipo `float`. No cômputo geral, há um ganho de desempenho na ordem de 14% quando comparado à máquina J9 sem JIT, que é a segunda mais rápida. Para nós este foi o resultado mais importante, pois justifica a adoção da especificação aqui proposta.

O tempo de execução de VERA em relação à máquina virtual SuperWaba foi cerca de 55% menor. Podemos inferir que esse ganho decorre basicamente da utilização de registradores, pois o compilador utilizado no Dell Axim suporta apenas `switch` na seleção da instrução. Nada podemos afirmar sobre o J9, pois seu código fonte é proprietário.

---

[12] YIHUEY (2004) conclui que nem sempre o JIT do J9 traz benefícios para o desempenho, pois existe um acréscimo de processamento que não pode ser desprezado.
[13] Vale lembrar que este foi um dos argumentos usados para justificar a criação da nova especificação.

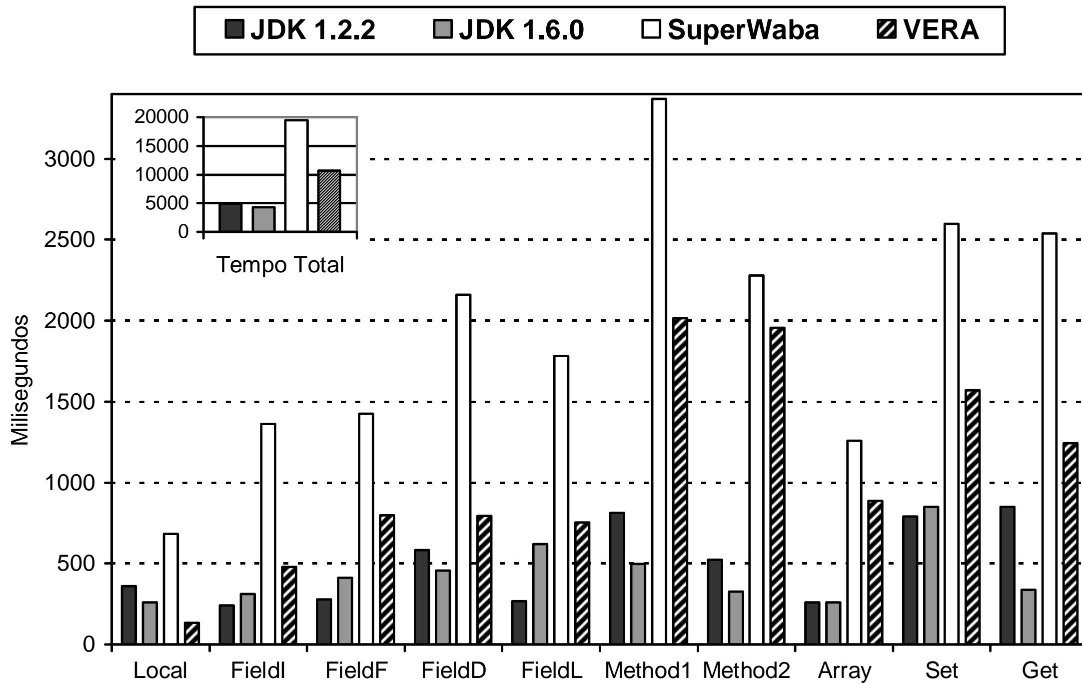

Figura 5: Gráfico comparativo com os tempos obtidos no Pentium 4-M

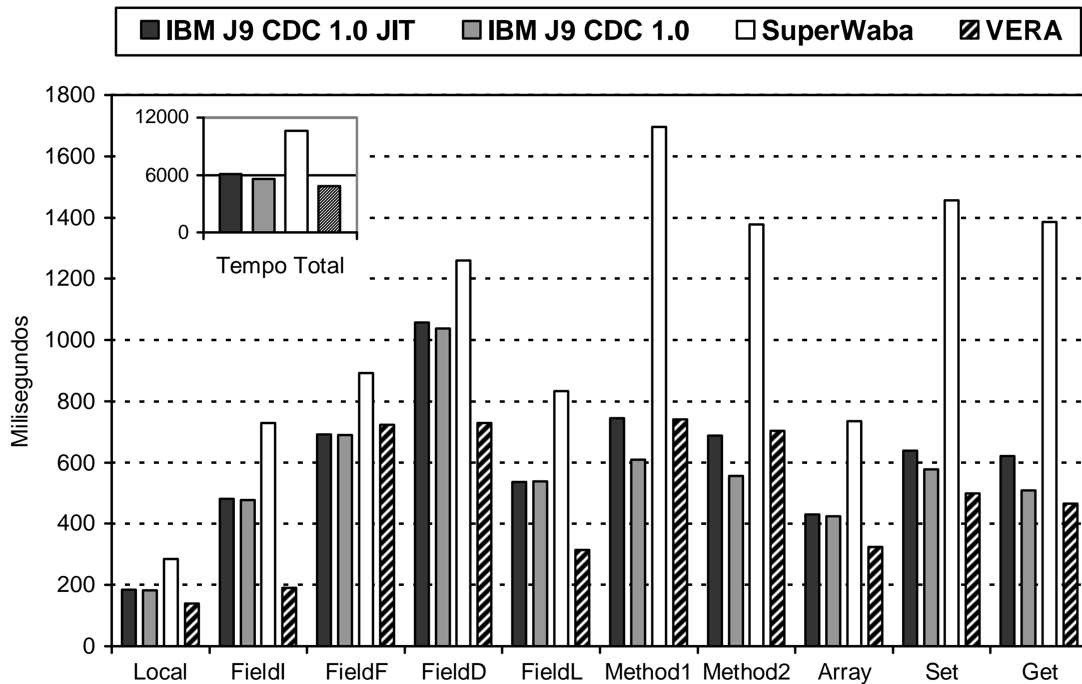

Figura 6: Gráfico comparativo com os tempos obtidos no Dell Axim X3

	JDK 1.2.2	JDK 1.6.0	SuperWaba	VERA
TestLocal	359	261	684	134
TestFieldI	240	310	1.364	478
TestFieldF	280	411	1.426	798
TestFieldD	581	458	2.159	795
TestFieldL	266	619	1.783	753
TestMethod1	813	499	3.369	2.015
TestMethod2	523	326	2.279	1.957
TestArray	260	261	1.259	887
TestSet	791	851	2.600	1.570
TestGet	851	338	2.539	1.242
Total	4.965	4.334	19.464	10.629

Tabela 29: Tempos em milisegundos dos testes realizados no Pentium 4-M

	Local	FieldI	FieldF	FieldD	FieldL	Method1	Method2	Array	Set	Get
VERA	1	2	3	3	3	1	1	4	1	1
JDK	1	6	6	6	6	6	6	7	3	3

Tabela 30: Número de instruções dentro do laço

	IBM J9 JIT	IBM J9	SuperWaba	VERA
TestLocal	185	182	284	140
TestFieldI	482	477	729	191
TestFieldF	691	690	892	724
TestFieldD	1.057	1.038	1.259	730
TestFieldL	536	538	834	315
TestMethod1	745	610	1.695	740
TestMethod2	687	557	1.378	703
TestArray	430	425	734	325
TestSet	639	577	1.457	500
TestGet	620	508	1.385	466
Total	6.072	5.602	10.647	4.834

Tabela 31: Tempos em milisegundos dos testes realizados no Dell Axim X3

4.3. Tamanho do código gerado

Mostraremos a seguir o tamanho de código gerado pelo JavaC do JDK 1.2.2 e 1.6.0, e também pelo compilador VERA.

A figura 7 exibe um gráfico com o tamanho do código gerado pelos compiladores para o programa Java que contém todos os testes. Para melhorar a comparação entre as plataformas, retiramos do programa compilado o construtor e os métodos `getTimeStamp` e `print`. Pelo gráfico verificamos que o JDK 1.6.0 gera um código 17,5% maior[14] que o 1.2.2. Já o compilador VERA gerou código 39% menor que o JDK 1.6.0 e 28% que o JDK 1.2.2. A figura 8 mostra um gráfico onde separamos os arquivos compilados em três partes: a parte das instruções, da tabela de constantes, e o restante, composto pelas definições de campos, métodos e classes. Por este gráfico, vemos que as instruções aumentaram o arquivo gerado por VERA em cerca de 10%, enquanto que a tabela de constantes caiu de 56% a 59%. Vemos também que a parte gasta com definições dos campos, métodos e da classe também caiu bastante em relação a Java, variando de 24% a 45%. Portanto, a economia gerada pelas mudanças na tabela de constantes, descritas na seção 3.5, tiveram uma grande importância na redução do tamanho do arquivo gerado por VERA, compensando em muito o acréscimo que o tamanho das instruções provoca.

Uma dessas mudanças foi o compartilhamento da tabela de constantes entre diversos arquivos. Para medir a redução no tamanho devido ao compartilhamento, escolhemos 6 pequenas classes, e deixamos apenas o protótipo dos métodos, removendo o código e retornando `0` ou `null` quando necessário. Isso fez com que o código gerado pelo 1.6.0 ficasse com o mesmo tamanho do 1.2.2. Além disso, mantivemos também os campos, e trocamos para `Object` todas as referências para outros tipos de objeto, pois o compilador VERA ainda não suporta referências circulares (classe A usa classe B que usa classe A). A figura 9 mostra os tamanhos obtidos com o compilador do JDK 1.2.2/1.6.0, e pelo compilador VERA com o compartilhamento ativado e desativado.

Apesar do ganho obtido com o compartilhamento ter sido pequeno, pois as classes não tinham nenhuma ligação entre si, acreditamos que a compilação de

---

[14] Analisando os arquivos gerados pelos dois compiladores, vemos que o 1.6.0 tem a tabela de constantes um pouco maior, e também adicionou 110 bytes com atributos para os métodos. O código em si mudou praticamente nada.

classes de um mesmo pacote possibilitará uma redução significativa, pois em geral essas classes se usam mutuamente, e haverá apenas uma referência para cada classe do pacote na tabela de constantes.

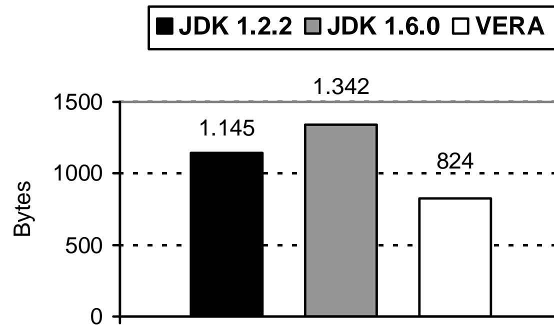

Figura 7: Tamanho do código gerado pela compilação do teste de desempenho
Foram removidos o construtor e os métodos `getTimeStamp` e `print`.

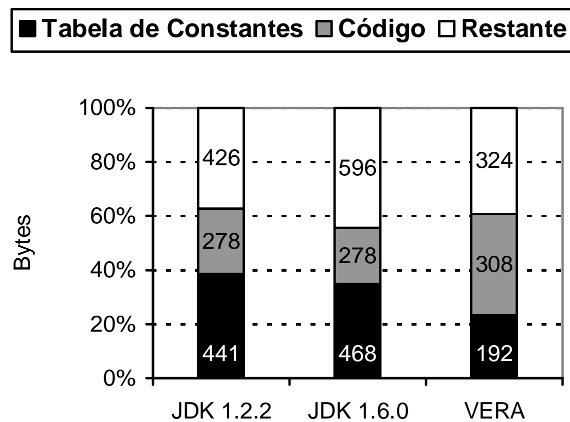

Figura 8: Composição do arquivo gerado pela compilação do teste de desempenho
Foram removidos o construtor e os métodos `getTimeStamp` e `print`.

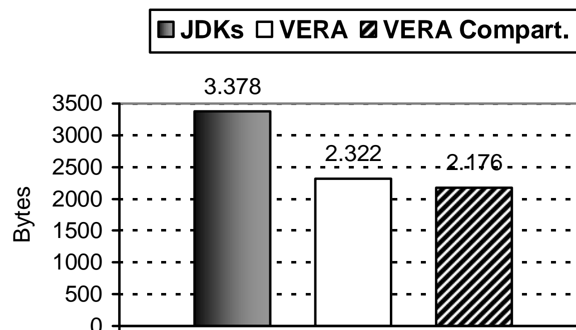

Figura 9: Tamanho do código gerado pela compilação dos protótipos

5 Conclusão

Nosso objetivo foi desenvolver uma especificação que favorecesse o desempenho sem a necessidade do uso de compilação sob demanda (JIT). Com base em artigos que diziam ter melhorado o desempenho da máquina virtual Java através da conversão dos bytecodes baseados em pilha para registradores, decidimos implementar uma nova especificação de bytecodes baseados em registradores. Era essencial que todos os tipos de operandos, como campos de instância e vetores, pudessem ser movidos para registradores, operados neles, e movidos de volta. Para definir quais instruções teriam um pouco mais de liberdade para operar diretamente com campos, vetores e constantes numéricas, escrevemos um interpretador virtual que coletou estatísticas de uso desses operandos em cerca de 14.000 classes. Ele identificou que, dentre as instruções aritméticas e lógicas, a que adiciona uma constante em um registrador, guardando o resultado em outro registrador aparece em pelo menos um quinto do total dessas instruções. Ele também identificou que a operação de desvio condicional mais freqüente foi a que verifica se um objeto é nulo. Decidimos então implementar na especificação as instruções mais freqüentes dos testes realizados.

Os programas usados em PDAs trabalham pouco com ponto flutuante, e o tipo `double` é o mais usado nos casos em que ponto flutuante é requerido. O tipo `float` ocorreu em apenas 0,8% das instruções, 16 vezes menos que o `double`. Como o `float` pode ser convertido para double, sem perda de precisão e com pouca perda de desempenho, decidimos eliminá-lo da especificação.

Estipulamos um total de 64 registradores para cada um dos tipos `int`, `Object`, `double` e `long`. Nas análises que fizemos apenas 2 métodos em quase 128 mil usavam mais do que 64 variáveis locais, portanto o limite de 64 registradores é aceitável.

Fizemos modificações também na tabela de constantes em relação ao arquivo `class` de Java. As constantes foram agrupadas por tipos, sendo que as descrições dos métodos, com seu nome, tipos de parâmetros e retorno, são desmembradas em referências para a tabela de constantes. Além disso, há a

possibilidade de compartilhar a tabela de constantes entre um conjunto de classes, reduzindo ainda o tamanho total.

Escolhemos a arquitetura *little-endian* para armazenar os tipos de dados. Além disso, as instruções serão codificadas em palavras de 32 bits, garantindo o alinhamento de forma que a leitura poderá ser feita de uma vez em arquiteturas little-endian.

Nós criamos um compilador e uma máquina virtual de referência, que denominamos VERA, para avaliar a eficiência da especificação. Fizemos testes de desempenho em um *notebook* Pentium 4-M e em um Pocket PC Dell Axim X3 com o uso de variáveis locais, campos de instância, vetores e chamadas à métodos. Os resultados de VERA foram comparados com os obtidos na máquina virtual do JDK 1.2.2, do JDK 1.6.0, ambas com o JIT desativado. Incluímos também a máquina virtual SuperWaba, e a IBM J9 CDC 1.0 para Pocket PC, que são utilizados em PDAs.

O resultado dos testes no Pentium 4-M demonstrou que VERA demorou o dobro do tempo em relação aos JDKs e a metade em relação ao SuperWaba. Em Ertl et al. (2005), foi obtida uma melhora de 32,3% da máquina de registradores proposta em relação à uma máquina JVM padrão, também em um Pentium 4 (no artigo não é informada que máquina virtual é essa, portanto não pudemos usá-la nos testes). Já em relação ao SuperWaba, VERA foi sempre mais rápido. Analisamos o código fonte disponível para o JDK 1.6.0, mas não encontramos uma razão que explicasse a diferença de velocidade. Por outro lado, conhecemos bem o código fonte da máquina virtual SuperWaba, e podemos inferir que o uso de registradores melhorou em pelo menos 27% o desempenho quando comparado ao uso de pilha.

Por outro lado, executando o teste no Pocket PC, que é um dos dispositivos para o qual a especificação foi projetada, VERA conseguiu uma redução de 14% no tempo de execução em comparação com o IBM J9, e uma redução de 55% em comparação com o SuperWaba. Notamos também um comportamento estranho do IBM J9, onde o uso do JIT gerou um código mais lento que sua não utilização. Acreditamos que isso se deve ao tempo da compilação ter excedido os ganhos obtidos com o aumento da velocidade de execução do código.

Fizemos também uma comparação no tamanho do código gerado pelos compiladores do JDK 1.2.2 e 1.6.0 com o gerado pelo compilador VERA. O resultado mostrou que VERA gera um arquivo compilado de 28% a 39% menor. Esse ganho deve-se às mudanças na tabela de constantes e nas definições das estruturas (classe, métodos e campos). Em VERA houve um aumento de cerca

de 10% no tamanho do código, enquanto que em Ertl et al. (2005), o acréscimo foi de 25%. O JDK 1.6.0 gerou um código maior que o 1.2.2 porque a tabela de constantes aumentou um pouco e diversos atributos foram inseridos em cada método.

Efetuamos um teste ativando o compartilhamento da tabela de constantes entre programas pelo compilador VERA. Como ele não suporta ainda referências circulares, escolhemos programas que não tinham relação entre si, e obtivemos uma redução de apenas 6,7%. Acreditamos que o compartilhamento trará uma redução maior com a compilação de classes pertencentes ao mesmo pacote.

Por fim, concluímos que a especificação atendeu às expectativas em relação ao aumento da velocidade e as superou no quesito tamanho do código gerado. Alguns ajustes poderão ser feitos na lista de instruções à medida que forem notados gargalos na execução dos programas, causados pela impossibilidade de operar diretamente certos tipos de operandos.

6
Referências Bibliográficas

ANTONIOLI D.N.; PILZ, M. **Analysis of the Java Class File Format**. Technical Report ifi-98.04, Department of Computer Science, University of Zurich, 1998.

BELL, J. R. **Threaded code**. Commun ACM, 16(6): 370-372, 2000.

CRAIG, I. D. **Virtual Machines**, Londres: Springer-Verlag, 2005, p. 157.

DALY, C. et al. **Platform Independent Dynamic Java Virtual Machine Analysis**: the Java Grande Forum Benchmark Suite. In: Joint ACM Java Grande - ISCOPE 2001 Conference, Stanford University, USA, 2001.

DAVIS, B. et al. **The case for virtual register machines**, In IVME '03: Proceedings of the 2003 workshop on Interpreters, Virtual machines and emulators, San Diego: ACM Press, p. 41-49, 2003.

DONOGHUE, D. et al. **Bigram analysis of Java bytecode sequences**, In PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, National University of Ireland, p. 187-192, 2002.

DONOGHUE, D.; POWER, J. F. **Identifying and Evaluating a Generic Set of Superinstructions for Embedded Java Programs**, In ESA/VLSI, CSREA Press, p. 192-198, 2004.

DOWLING, T.; POWER, J. F.; WALDRON, J. **Relating Static and Dynamic Measurements for the Java Virtual Machine Instruction Set**. In Symposium on Mathematical Methods and Computational Techniques in Electronic Engineering, Athens, Greece, 2001.

ERTL, M. A.; GREGG, D. **The behaviour of efficient virtual machine interpreters on modern architectures**. In Euro-Par 2001, pages 403–412. Springer LNCS 2150, 2001.

ERTL, M. A.; GREGG, D. **Combining stack caching with dynamic superinstructions**, In IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators, Washington, D.C.: ACM Press, p. 7-14, 2004.

ERTL, A. et al. **Virtual machine showdown**: stack versus registers, in Proceedings of the ACM/SIGPLAN Conference of Virtual Execution Environments (VEE 05), p. 153-163, Chicago Illinois, 2005.

HSIEH, C. A.; GYLLENHAAL, J. C.; HWU, W. **Java bytecode to native code translation**: the caffeine prototype and preliminary results, In MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, p. 90-99, Paris, France, 1996.

PORTHOUSE, C. **Jazelle**: High performance Java on embedded devices, ARM Ltd.

PUGH, W. **Compressing Java class files**. In PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, ACM Press, p. 247-258, Atlanta, GE, 1999.

RAYSIDE, D.; MAMAS, E.; HONS E. **Compact Java binaries for embedded systems**, In CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, p. 9-22, Mississauga, Canada, 1999.

STEPHENSON, B.; HOLST W. **A quantitative analysis of Java bytecode sequences**, In PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java, Trinity College Dublin, p. 15-20, Las Vegas, NV, 2004.

SULLIVAN, G. T. et al. **Dynamic native optimization of interpreters**, In IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators, ACM Press, p. 50-57, San Diego, CA, 2003.

WALDRON, J. **Analysis of Virtual Machine Stack Frame Usage by Java Methods**. Internet, Multimedia Systems and Applications, p. 271-274, 1999.

WALDRON, J. et al. **Comparison of factors influencing bytecode usage in the Java Virtual Machine**. In Second International Conference and Exhibition on the Practical Application of Java, p. 315-327, Manchester, UK, 2000.

YIHUEY, LI **Runrime Performance Evaluation of Junst-In-Time Compiler Enabled J9 Virtual Machine**, Arizona State University East, AZ, 2004.

IGOR PAVLOV, **7Zip Compression Tool**, http://www.sevenzip.org, 1999

BCEL - **Byte Code Engineering Library**, http://jakarta.apache.org/bcel.

SUPERWABA - **Virtual Machine for PDAs**, http://www.superwaba.com.br

IBM J9 - **WebSphere Everyplace Micro Environment**

http://www.ibm.com/developerworks/websphere/zones/wireless/weme_eval_runtimes.html

7
Apêndice

Neste apêndice colocamos tabelas que complementam as informações dos capítulos anteriores. Na seção 7.1 temos a lista dos bytecodes Java. Na 7.2, o mapeamento desses bytecodes nas instruções que criamos. Por fim, na seção 7.3, listamos o código gerado pelos compiladores VERA e JavaC do JDK 1.2.2.

7.1. Lista dos bytecodes Java

A tabela 32 exibe os bytecodes Java na ordem definida pelo JCP, seguindo uma categorização que criamos.

CATEGORIA	Movimentação e Conversão	Aritméticos e Lógicos	Desvios	Outros

	0	35	70	105	140	175
0	NOP	FLOAD_1	FSTORE_3	LMUL	F2L	DRETURN
1	ACONST_NULL	FLOAD_2	DSTORE_0	FMUL	F2D	ARETURN
2	ICONST_M1	FLOAD_3	DSTORE_1	DMUL	D2I	RETURN
3	ICONST_0	DLOAD_0	DSTORE_2	IDIV	D2L	GETSTATIC
4	ICONST_1	DLOAD_1	DSTORE_3	LDIV	D2F	PUTSTATIC
5	ICONST_2	DLOAD_2	ASTORE_0	FDIV	I2B	GETFIELD
6	ICONST_3	DLOAD_3	ASTORE_1	DDIV	I2C	PUTFIELD
7	ICONST_4	ALOAD_0	ASTORE_2	IREM	I2S	INVOKEVIRTUAL
8	ICONST_5	ALOAD_1	ASTORE_3	LREM	LCMP	INVOKESPECIAL
9	LCONST_0	ALOAD_2	IASTORE	FREM	FCMPL	INVOKESTATIC
10	LCONST_1	ALOAD_3	LASTORE	DREM	FCMPG	INVOKEINTERFACE
11	FCONST_0	IALOAD	FASTORE	INEG	DCMPL	
12	FCONST_1	LALOAD	DASTORE	LNEG	DCMPG	NEW
13	FCONST_2	FALOAD	AASTORE	FNEG	IFEQ	NEWARRAY
14	DCONST_0	DALOAD	BASTORE	DNEG	IFNE	ANEWARRAY
15	DCONST_1	AALOAD	CASTORE	ISHL	IFLT	ARRAYLENGTH
16	BIPUSH	BALOAD	SASTORE	LSHL	IFGE	ATHROW
17	SIPUSH	CALOAD	POP	ISHR	IFGT	CHECKCAST
18	LDC	SALOAD	POP2	LSHR	IFLE	INSTANCEOF
19	LDC_W	ISTORE	DUP	IUSHR	IF_ICMPEQ	MONITORENTER
20	LDC2_W	LSTORE	DUP_X1	LUSHR	IF_ICMPNE	MONITOREXIT
21	ILOAD	FSTORE	DUP_X2	IAND	IF_ICMPLT	WIDE
22	LLOAD	DSTORE	DUP2	LAND	IF_ICMPGE	MULTIANEWARRAY
23	FLOAD	ASTORE	DUP2_X1	IOR	IF_ICMPGT	IFNULL
24	DLOAD	ISTORE_0	DUP2_X2	LOR	IF_ICMPLE	IFNONNULL
25	ALOAD	ISTORE_1	SWAP	IXOR	IF_ACMPEQ	GOTO_W
26	ILOAD_0	ISTORE_2	IADD	LXOR	IF_ACMPNE	JSR_W
27	ILOAD_1	ISTORE_3	LADD	IINC	GOTO	BREAKPOINT
28	ILOAD_2	LSTORE_0	FADD	I2L	JSR	
29	ILOAD_3	LSTORE_1	DADD	I2F	RET	
30	LLOAD_0	LSTORE_2	ISUB	I2D	TABLESWITCH	
31	LLOAD_1	LSTORE_3	LSUB	L2I	LOOKUPSWITCH	
32	LLOAD_2	FSTORE_0	FSUB	L2F	IRETURN	
33	LLOAD_3	FSTORE_1	DSUB	L2D	LRETURN	
34	FLOAD_0	FSTORE_2	IMUL	F2I	FRETURN	

Tabela 32: Bytecodes Java

7.2. Mapeamento entre os bytecodes Java e as novas instruções

Na tabela 33 exibimos um mapeamento de todos os bytecodes da linguagem Java com o equivalente na nova especificação. Desta forma, garantiremos que ela cobre a linguagem Java.

De maneira geral, as conversões são feitas de forma direta: `load` e `store` são convertidas em `mov`, as operações aritméticas e lógicas são convertidas para a operação correspondente, usando registradores como operandos. As operações com float foram ignoradas, pois o tipo float foi eliminado, como já visto.

Instruções que apenas manipulam a pilha foram marcadas como n/a (não aplicável). A instrução `nop` não foi implementada.

As instruções `jsr/ret` não foram implementadas, pois o suporte a `finally` será implementado de outra forma, colocando o conteúdo *inlined*. Muitas vezes o `finally` é usado de forma incorreta. Por exemplo: `try {a} catch {b} finally {c}` equivale a `try {a} catch {b} {c}`, que não requer o `jsr/ret` para ser implementado. O único caso em que ele é necessário é quando usado sem o `catch`: `try {a} finally {c}`.

Bytecode Java	Novas instruções
AALOAD	mov regO, regO[regI]
AASTORE	mov regO[regI], regO
ACONST_NULL	mov regO, null
ALOAD	mov regO, regO
ALOAD_0	mov regO, regO
ALOAD_1	mov regO, regO
ALOAD_2	mov regO, regO
ALOAD_3	mov regO, regO
ANEWARRAY	newArray
ARETURN	return reg
ARRAYLENGTH	mov regI, regO.tamvet
ASTORE	mov regO, regO
ASTORE_0	mov regO, regO
ASTORE_1	mov regO, regO
ASTORE_2	mov regO, regO
ASTORE_3	mov regO, regO

Bytecode Java	Novas instruções
ICONST_4	mov regI, 4
ICONST_5	mov regI, 5
ICONST_M1	mov regI, -1
IDIV	div regI, regI, regI
IF_ACMPEQ	jeq regO, regO, delta
IF_ACMPNE	jne regO, regO, delta
IF_ICMPEQ	jeq regI, regI, delta
IF_ICMPGE	jge regI, regI, delta
IF_ICMPGT	jgt regI, regI, delta
IF_ICMPLE	jle regI, regI, delta
IF_ICMPLT	jlt regI, regI, delta
IF_ICMPNE	jne regI, regI, delta
IFEQ	jeq reg?, 0, delta
IFGE	jge reg?, 0, delta
IFGT	jgt reg?, 0, delta
IFLE	jle reg?, 0, delta

Bytecode Java	Novas instruções	Bytecode Java	Novas instruções
ATHROW	throw	IFLT	jlt reg?, 0, delta
BALOAD	mov reglb, regO[regl]	IFNE	jne reg?, 0, delta
BASTORE	mov regO[regl], reglb	IFNONNULL	jne regO, 0
BIPUSH	mov regl, valor	IFNULL	jeq regO, 0
BREAKPOINT	break	IINC	add regl, valor
CALOAD	mov reglc, regO[regl]	ILOAD	mov regl, regl
CASTORE	mov regO[regl], reglc	ILOAD_0	mov regl, regl
CHECKCAST	checkcast	ILOAD_1	mov regl, regl
D2F	n/a	ILOAD_2	mov regl, regl
D2I	D2I regl, regD	ILOAD_3	mov regl, regl
D2L	D2L regL, regD	IMUL	mul regl, regl, regl
DADD	add regD, regD, regD	INEG	sub regl, 0, regl
DALOAD	mov regO[regl], regD	INSTANCEOF	instanceof
DASTORE	mov regD, regO[regl]	INVOKEINTERFACE	CallExternal
DCMPG	JGT regD, regD, delta	INVOKESPECIAL	CallInternal
DCMPL	JLT regD, regD, delta	INVOKESTATIC	CallInternal/CallExternal
DCONST_0	mov regD, 0	INVOKEVIRTUAL	CallInternal/CallExternal
DCONST_1	mov regD, 1	IOR	or regl, regl, regl
DDIV	div regD, regD, regD	IREM	mod regl, regl, regl
DLOAD	mov regD, regD	IRETURN	ret regl
DLOAD_0	mov regD, regD	ISHL	shl regl, regl, regl
DLOAD_1	mov regD, regD	ISHR	sshr regl, regl, regl
DLOAD_2	mov regD, regD	ISTORE	mov regl, regl
DLOAD_3	mov regD, regD	ISTORE_0	mov regl, regl
DMUL	mul regD, regD, regD	ISTORE_1	mov regl, regl
DNEG	mov regD,0; sub regD, regD(=0), regD	ISTORE_2	mov regl, regl
DREM	mod regD, regD, regD	ISTORE_3	mov regl, regl
DRETURN	ret regD	ISUB	sub regl, regl, regl
DSTORE	mov regD, regD	IUSHR	ushr regl, regl, regl
DSTORE_0	mov regD, regD	IXOR	xor regl, regl, regl
DSTORE_1	mov regD, regD	JSR	***
DSTORE_2	mov regD, regD	JSR_W	***
DSTORE_3	mov regD, regD	L2D	L2D regD, regL
DSUB	sub regD, regD, regD	L2F	n/a
DUP	n/a	L2I	L2I regL, regl

Bytecode Java	Novas instruções
DUP_X1	n/a
DUP_X2	n/a
DUP2	n/a
DUP2_X1	n/a
DUP2_X2	n/a
F2D	n/a
F2I	n/a
F2L	n/a
FADD	n/a
FALOAD	n/a
FASTORE	n/a
FCMPG	n/a
FCMPL	n/a
FCONST_0	n/a
FCONST_1	n/a
FCONST_2	n/a
FDIV	n/a
FLOAD	n/a
FLOAD_0	n/a
FLOAD_1	n/a
FLOAD_2	n/a
FLOAD_3	n/a
FMUL	n/a
FNEG	n/a
FREM	n/a
FRETURN	n/a
FSTORE	n/a
FSTORE_0	n/a
FSTORE_1	n/a
FSTORE_2	n/a
FSTORE_3	n/a
FSUB	n/a
GETFIELD	mov reg?, [cpIntr,cpExtr]
GETSTATIC	mov reg?, [estInt,estExtr]

Bytecode Java	Novas instruções
LADD	add regL, regL, regL
LALOAD	mov regL, regO[regI]
LAND	and regL, regL, regL
LASTORE	mov regO[regI], regL
LCMP	j<cond> regL, regL, delta
LCONST_0	mov regL, 0
LCONST_1	mov regL, 1
LDC	mov regI, simb
LDC_W	mov regI, simb
LDC2_W	mov regL, simb mov regD, simb
LDIV	div regL, regL, regL
LLOAD	mov regL, regL
LLOAD_0	mov regL, regL
LLOAD_1	mov regL, regL
LLOAD_2	mov regL, regL
LLOAD_3	mov regL, regL
LMUL	mul regL, regL, regL
LNEG	mov regL,0; sub regL, regL(0), regL
LOOKUPSWITCH	switch
LOR	or regL, regL, regL
LREM	mod regL, regL, regL
LRETURN	ret regL
LSHL	shl regL, regL, regL
LSHR	sshr regL, regL, regL
LSTORE	mov regL, regL
LSTORE_0	mov regL, regL
LSTORE_1	mov regL, regL
LSTORE_2	mov regL, regL
LSTORE_3	mov regL, regL
LSUB	sub regL, regL, regL
LUSHR	ushr regL, regL, regL
LXOR	xor regL, regL, regL
MONITORENTER	syncStart
MONITOREXIT	syncEnd

Bytecode Java	Novas instruções	Bytecode Java	Novas instruções
GOTO	jmp s24	MULTIANEWARRAY	vários new array
GOTO_W	jmp s24	NEW	new obj
I2B	I2B regI, regI	NEWARRAY	new array
I2C	I2C regI, regI	NOP	***
I2D	I2D regD, regI	POP	n/a
I2F	n/a	POP2	n/a
I2L	I2L regL, regI	PUTFIELD	mov [cpIntr,cpExtr], reg?
I2S	I2S regI, regI	PUTSTATIC	mov [estIntr,estExtr], reg?
IADD	add regI, regI, regI	RET	***
IALOAD	mov regI, regO[regI]	RETURN	return
IAND	and regI, regI, regI	SALOAD	mov regIs, regO[regI]
IASTORE	mov regO[regI], regI	SASTORE	mov regO[regI], regIs
ICONST_0	mov regI, 0	SIPUSH	mov regI, valor
ICONST_1	mov regI, 1	SWAP	n/a
ICONST_2	mov regI, 2	TABLESWITCH	switch
ICONST_3	mov regI, 3	WIDE	n/a

Tabela 33: Conversão entre bytecodes Java e as novas instruções.

***: não existe instrução correspondente   n/a: não aplicável

7.3.
Listagem do código gerado pelos compiladores

A tabela a seguir mostra a listagem do código gerado pelo compilador VERA e pelo compilador do JDK 1.2.2, excluindo o construtor e os métodos `print` e `getTimeStamp`. Em primeiro lugar, são exibidos os campos da classe, em seguida as informações dos métodos com os códigos, e por fim, os dados contidos nas tabelas de constantes.

VERA	**JDK 1.2.2**

Class Name : AllTests
Source File: ./AllTests
AccessFlag : Public InfoIsHere
Instance Fields (I32):
   Name: 4097 - fieldI
   Type: 6
   AccessFlag: Private
   Value: 0
 	
   Name: 4101 - age
   Type: 6
   AccessFlag: Private
   Value: 0
 	
Instance Fields (I64):
   Name: 4100 - fieldL
   AccessFlag: Private
   Value: 0
 	
Instance Fields (Dbl):
   Name: 4098 - fieldF
   AccessFlag: Private
   Value: 0.000000
 	
   Name: 4099 - fieldD
   AccessFlag: Private
   Value: 0.000000
 	
Methods:
	Decompiled by Decafe PRO
	Classes: 1
	Methods: 17
	Fields: 5
	
	private int fieldI;
	private float fieldF;
	private double fieldD;
	private long fieldL;
	private int age;

Name: 4102 - testLocal
   AccessFlag: Public
   Parameters Count: 0
   Return Type: 0 -
   How many registers:
      iCount: 2
   Code:
  int j = 0;
1 MOV_regI_s18 : 0 <- #0
  for (int i=10000000; i > 0; i--)
2 MOV_regI_sym : 1 <- sym#1
3 JUMP_s24 : -> 2
      j += 2;
4 INC_regI : 0 <- 0 #2
5 DECJGTZ_regI : 1 -> -1
6 RETURN_void :
	
	    int i = 0;
	1 0:iconst_0
	2 1:istore_1
	    for(int j = 0x989680; j > 0; j-
	-)
	3 2:ldc1 #1 <Int 0x989680>
	4 4:istore_2
	5 5:goto 14
	        i += 2;
	6 8:iinc 1 2
	7 11:iinc 2 -1
	8 14:iload_2
	9 15:ifgt 8
	10 18:return

VERA	**JDK 1.2.2**

Name: 4103 - testFieldI
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 3
Code:
int c = 2;
1 MOV_regI_s18 : 0 <- #2
for (int i=10000000; i > 0; i--)
2 MOV_regI_sym : 1 <- sym#1
3 JUMP_s24 : -> 3
fieldI += c;
4 MOV_regI_thisField : 2 <- 0
5 ADD_thisField_regI_regI :
6 DECJGTZ_regI : 1 -> -2
7 RETURN_void :
	byte byte0 = 2;
	1 0:iconst_2
	2 1:istore_1
	for(int i = 0x989680; i > 0; i--)
	3 2:ldc1 #1 <Int 0x989680>
	4 4:istore_2
	5 5:goto 21
	fieldI += byte0;
	6 8:aload_0
	7 9:dup
	8 10:getfield #8 <int fieldI>
	9 13:iload_1
	10 14:iadd
	11 15:putfield #8 <int fieldI>
	12 18:iinc 2 -1
	13 21:iload_2
	14 22:ifgt 8
	15 25:return

Name: 4104 - testFieldF
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 1
dCount: 3
Code:
float c = 2;
1 MOV_regD_s18 : 0 <- #2
for (int i=10000000; i > 0; i--)
2 MOV_regI_sym : 0 <- sym#1
3 JUMP_s24 : -> 4
fieldF += c;
4 MOV_regD_thisField : 2 <- 1
5 ADD_regD_regD_regD : 1 <- 2 0
6 MOV_thisField_regD : 1 <- 1
7 DECJGTZ_regI : 0 -> -3
8 RETURN_void :
	float f = 2.0F;
	1 0:fconst_2
	2 1:fstore_1
	for(int i = 0x989680; i > 0; i--)
	3 2:ldc1 #1 <Int 0x989680>
	4 4:istore_2
	5 5:goto 21
	fieldF += f;
	6 8:aload_0
	7 9:dup
	8 10:getfield #7 <float fieldF>
	9 13:fload_1
	10 14:fadd
	11 15:putfield #7 <float fieldF>
	12 18:iinc 2 -1
	13 21:iload_2
	14 22:ifgt 8
	15 25:return

Name: 4105 - testFieldD
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 1
dCount: 3
Code:
double c = 2;
1 MOV_regD_s18 : 0 <- #2
for (int i=10000000; i > 0; i--)
2 MOV_regI_sym : 0 <- sym#1
3 JUMP_s24 : -> 4
fieldD += c;
4 MOV_regD_thisField : 2 <- 2
5 ADD_regD_regD_regD : 1 <- 2 0
6 MOV_thisField_regD : 2 <- 1
7 DECJGTZ_regI : 0 -> -3
8 RETURN_void :
	double d = 2D;
	1 0:ldc2w #17 <Double 2D>
	2 3:dstore_1
	for(int i = 0x989680; i > 0; i--)
	3 4:ldc1 #1 <Int 0x989680>
	4 6:istore_3
	5 7:goto 23
	fieldD += d;
	6 10:aload_0
	7 11:dup
	8 12:getfield #6 <double fieldD>
	9 15:dload_1
	10 16:dadd
	11 17:putfield #6 <double fieldD>
	12 20:iinc 3 -1
	13 23:iload_3
	14 24:ifgt 10
	15 27:return

Name: 4106 - testFieldL
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 1
lCount: 3
	long l = 2L;
	1 0:ldc2w #13 <Long 2L>
	2 3:lstore_1
	for(int i = 0x989680; i > 0; i--)
	3 4:ldc1 #1 <Int 0x989680>
	4 6:istore_3
	5 7:goto 23

VERA	**JDK 1.2.2**

Code:
long c = 2;
1 MOV_regL_s18 : 0 <- #2
for (int i=10000000; i > 0; i--)
2 MOV_regI_sym : 0 <- sym#1
3 JUMP_s24 : -> 4
fieldL += c;
4 MOV_regL_thisField : 2 <- 3
5 ADD_regL_regL_regL : 1 <- 2 0
6 MOV_thisField_regL : 3 <- 1
7 DECJGTZ_regI : 0 -> -3
8 RETURN_void :
	fieldL += l;
	6 10:aload_0
	7 11:dup
	8 12:getfield #9 <long fieldL>
	9 15:lload_1
	10 16:ladd
	11 17:putfield #9 <long fieldL>
	12 20:iinc 3 -1
	13 23:iload_3
	14 24:ifgt 10
	15 27:return

Name: 4107 - testMethod1
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 1
Code:
for (int i=10000000; i > 0; i--)
1 MOV_regI_sym : 0 <- sym#1
2 JUMP_s24 : -> 3
method(50,30,150,true);
3 CALL_internal : 8-> 0 par: 51
4 DECJGTZ_regI : 0 -> -2
5 RETURN_void :
	for(int i = 0x989680; i > 0; i--)
	1 0:ldc1 #1 <Int 0x989680>
	2 2:istore_1
	3 3:goto 22
	method(50, 30D, 150L, true);
	4 6:aload_0
	5 7:bipush 50
	6 9:ldc2w #19 <Double 30D>
	7 12:ldc2w #15 <Long 150L>
	8 15:iconst_1
	9 16:invokevirtual #11 <void method(int, double, long, boolean)>
	10 19:iinc 1 -1
	11 22:iload_1
	12 23:ifgt 6
	13 26:return

Name: 4108 - testMethod2
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 3
dCount: 1
lCount: 1
Code:
int i1 = 50;
1 MOV_regI_s18 : 0 <- #50
double d1 = 30;
2 MOV_regD_s18 : 0 <- #30
long l1 = 150;
3 MOV_regL_s18 : 0 <- #150
boolean b1 = true;
4 MOV_regI_s18 : 1 <- #1
for (int i=10000000; i > 0; i--)
5 MOV_regI_sym : 2 <- sym#1
6 JUMP_s24 : -> 3
method(i1,d1,l1,b1);
7 CALL_internal : 8 -> 0 par: 0
8 DECJGTZ_regI : 2 -> -2
9 RETURN_void :
	byte byte0 = 50;
	1 0:bipush 50
	2 2:istore_1
	double d = 30D;
	3 3:ldc2w #19 <Double 30D>
	4 6:dstore_2
	long l = 150L;
	5 7:ldc2w #15 <Long 150L>
	6 10:lstore 4
	boolean flag = true;
	7 12:iconst_1
	8 13:istore 6
	for(int i = 0x989680; i > 0; i--)
	9 15:ldc1 #1 <Int 0x989680>
	10 17:istore 7
	11 19:goto 35
	method(byte0, d, l, flag);
	12 22:aload_0
	13 23:iload_1
	14 24:dload_2
	15 25:lload 4
	16 27:iload 6
	17 29:invokevirtual #11 <void method(int,double,long, boolean)>
	18 32:iinc 7 -1
	19 35:iload 7
	20 37:ifgt 22
	21 40:return

Name: 4109 - testArray
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 4
oCount: 1
	int ai[] = new int[10];
	1 0:bipush 10
	2 2:newarray int[]
	3 4:astore_1
	for(int i = 0x989680; i > 0; i--)
	4 5:ldc1 #1 <Int 0x989680>

VERA	**JDK 1.2.2**

Code:
int[] a = new int[10];
1 NEWARRAY_len : 0 = sym#16, #10
for (int i=10000000; i > 0; i--)
2 MOV_regI_sym : 0 <- sym#1
3 JUMP_s24 : -> 5
a[5] += i;
4 MOV_regI_s18 : 1 <- #5
5 MOV_regI_aru : 3 <- 0[1]
6 ADD_regI_regI_regI : 2 <- 3 0
7 MOV_aru_regI : 0[1] <- 2
8 DECJGTZ_regI : 0 -> -4
9 RETURN_void :
	5 7:istore_2
	6 8:goto 21
	ai[5] += i;
	7 11:aload_1
	8 12:iconst_5
	9 13:dup2
	10 14:iaload
	11 15:iload_2
	12 16:iadd
	13 17:iastore
	14 18:iinc 2 -1
	15 21:iload_2
	16 22:ifgt 11
	17 25:return

Name: 4110 - method AccessFlag: Public Parameters Count: 4 Return Type: 0 - How many registers: iCount: 2 dCount: 1 lCount: 1 Index of Parameters: 6 9 7 2 Register Types of Parameters: 0 2 3 0 Code: 1 [177] RETURN_void :	1 0:return

Name: 4111 - testSet
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 1
Code:
for (int i=10000000; i > 0; i--)
1 MOV_regI_sym : 0 <- sym#1
2 JUMP_s24 : -> 2
setAge(10);
3 CALL_internal : 11-> 0 par: 11
4 DECJGTZ_regI : 0 -> -1
5 RETURN_void :
	for(int i = 0x989680; i > 0; i--)
	1 0:ldc1 #1 <Int 0x989680>
	2 2:istore_1
	3 3:goto 15
	setAge(10);
	4 6:aload_0
	5 7:bipush 10
	6 9:invokevirtual #12
	<Method void setAge(int)>
	7 12:iinc 1 -1
	8 15:iload_1
	9 16:ifgt 6
	10 19:return

Name: 4112 - testGet
AccessFlag: Public
Parameters Count: 0
Return Type: 0 -
How many registers:
iCount: 2
Code:
for (int i=10000000; i > 0; i--)
1 MOV_regI_sym : 1 <- sym#1
2 JUMP_s24 : -> 2
a = getAge();
3 CALL_internal : 12-> 0 ret: 0
4 DECJGTZ_regI : 1 -> -1
5 RETURN_void :
	for(int j = 0x989680; j > 0; j--
	)
	1 0:ldc1 #1 <Int 0x989680>
	2 2:istore_2
	3 3:goto 14
	{
	int i = getAge();
	4 6:aload_0
	5 7:invokevirtual #10
	<Method int getAge()>
	6 10:istore_1
	}
	7 11:iinc 2 -1
	8 14:iload_2
	9 15:ifgt 6
	10 18:return

Name: 4113 - setAge AccessFlag: Public Parameters Count: 1 Return Type: 0 - How many registers:	

VERA	JDK 1.2.2
iCount: 1 Index of Parameters: 6 Register Types of Parameters: 0 Code: age = a; 1 MOV_thisField_regI : 4 <- 0 2 RETURN_void :	age = i; 1 0:aload_0 2 1:iload_1 3 2:putfield #5 <int age> 4 5:return
Name: 4114 - getAge AccessFlag: Public Parameters Count: 0 Return Type: 6 - I How many registers: iCount: 1 Code: return age; 1 MOV_regI_thisField : 0 <- 4 2 RETURN_regI : 0	return age; 1 0:aload_0 2 1:getfield #5 <int age> 3 4:ireturn
ConstantPool: i32s: i32: 10000000 Class Idents: name: AllTests Method/Field Idents: name: fieldI name: fieldF name: fieldD name: fieldL name: age name: testLocal name: testFieldI name: testFieldF name: testFieldD name: testFieldL name: testMethod1 name: testMethod2 name: testArray name: method name: testSet name: testGet name: setAge name: getAge name: getTimeStamp name: print	1: INTEGER: int_value=10000000 2: CLASS: name=AllTests 3: CLASS: name=java/lang/Object 4: METHODREF: class=java/lang/Object, na- me=<init>, type=()V 5: FIELDREF: class=AllTests, name=age, type=I 6: FIELDREF: class=AllTests, name=fieldD, type=D 7: FIELDREF: class=AllTests, name=fieldF, type=F 8: FIELDREF: class=AllTests, name=fieldI, type=I 9: FIELDREF: class=AllTests, name=fieldL, type=J 10: METHODREF: class=AllTests, name=getAge, type=()I 11: METHODREF: class=AllTests, name=method, type=(IDJZ)V 12: METHODREF: class=AllTests, name=setAge, type=(I)V 13: LONG: long_value=2 15: LONG: long_value=150 17: DOUBLE: double_value=2.0 19: DOUBLE: double_value=30.0 21: NAMEANDTYPE: name=<init>, descriptor=()V 22: NAMEANDTYPE: name=age, descriptor=I 23: NAMEANDTYPE: name=fieldD, descriptor=D 24: NAMEANDTYPE: name=fieldF, descriptor=F 25: NAMEANDTYPE: name=fieldI, descriptor=I 26: NAMEANDTYPE: name=fieldL, descriptor=J 27: NAMEANDTYPE: name=getAge, descriptor=()I 28: NAMEANDTYPE: name=method, descriptor=(IDJZ)V 29: NAMEANDTYPE: name=setAge, descriptor=(I)V 30: UTF8: string=()I 31: UTF8: string=()V

VERA	**JDK 1.2.2**
	32: UTF8: string=(I)V 33: UTF8: string=(IDJZ)V 34: UTF8: string=<init> 35: UTF8: string=AllTests 36: UTF8: string=Code 37: UTF8: string=D 38: UTF8: string=F 39: UTF8: string=I 40: UTF8: string=J 41: UTF8: string=age 42: UTF8: string=fieldD 43: UTF8: string=fieldF 44: UTF8: string=fieldI 45: UTF8: string=fieldL 46: UTF8: string=getAge 47: UTF8: string=java/lang/Object 48: UTF8: string=method 49: UTF8: string=setAge 50: UTF8: string=testArray 51: UTF8: string=testFieldD 52: UTF8: string=testFieldF 53: UTF8: string=testFieldI 54: UTF8: string=testFieldL 55: UTF8: string=testGet 56: UTF8: string=testLocal 57: UTF8: string=testMethod1 58: UTF8: string=testMethod2 59: UTF8: string=testSet

Tabela 34: Listagem do código gerado pelos compiladores VERA e JDK 1.2.2.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo