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-

PUC-Rio - Certificação Digital Nº 0510973/CA

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

PUC-Rio - Certificação Digital Nº 0510973/CA

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

PUC-Rio - Certificação Digital Nº 0510973/CA

(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.

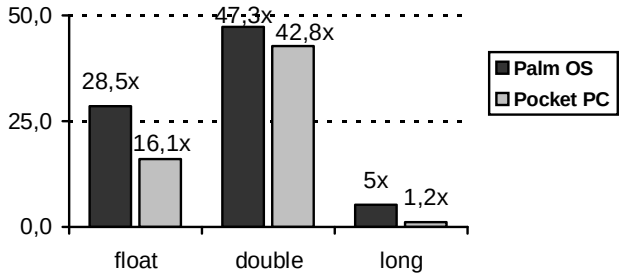

Figura 3: Comparação dos tipos primitivos em relação ao `int`.
Palm OS: Treo 650 com processador PXA270 XScale 312MHz.
Pocket PC: Dell Axim X3 com processador PXA270 XScale 300MHz.

```
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

PUC-Rio - Certificação Digital Nº 0510973/CA

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.

PUC-Rio - Certificação Digital Nº 0510973/CA

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-

PUC-Rio - Certificação Digital Nº 0510973/CA

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 {...}`

PUC-Rio - Certificação Digital Nº 0510973/CA

- 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

PUC-Rio - Certificação Digital Nº 0510973/CA

```
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

PUC-Rio - Certificação Digital Nº 0510973/CA

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.

PUC-Rio - Certificação Digital Nº 0510973/CA

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).

PUC-Rio - Certificação Digital Nº 0510973/CA

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`.

PUC-Rio - Certificação Digital Nº 0510973/CA

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).

PUC-Rio - Certificação Digital Nº 0510973/CA

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`.

PUC-Rio - Certificação Digital Nº 0510973/CA

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

PUC-Rio - Certificação Digital Nº 0510973/CA

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_reg0`. 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.

PUC-Rio - Certificação Digital Nº 0510973/CA

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.

PUC-Rio - Certificação Digital Nº 0510973/CA

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

PUC-Rio - Certificação Digital Nº 0510973/CA

#	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

PUC-Rio - Certificação Digital Nº 0510973/CA

#	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

#	Instr	Operação	Tipo
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

PUC-Rio - Certificação Digital Nº 0510973/CA

#	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.

PUC-Rio - Certificação Digital Nº 0510973/CA

#	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

PUC-Rio - Certificação Digital Nº 0510973/CA

#	**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

PUC-Rio - Certificação Digital Nº 0510973/CA