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.