Compressão de Byte-Code Utilizando Recuperação de Sintaxe Abstrata

Bruno Kraychete da Costa

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Compressão de Byte-Code Utilizando Recuperação de Sintaxe Abstrata

Bruno Kraychete da Costa[1]

Fevereiro de 2000

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (IC-UNICAMP) (Orientador)
- Profa. Dra. Edna Natividade da Silva Barros (DI-UFPE)
- Prof. Dr. Luiz Eduardo Buzato (IC-UNICAMP)
- Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP) (Suplente)

[1]**Apoio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico — CNPq (proc. 131.947/98-2).**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Costa, Bruno Kraychete da

C823c Compressão de byte-code utilizando recuperação de sintaxe abstrata / Bruno Kraychete da Costa -- Campinas, [S.P. :s.n.], 2000.

Orientador : Guido Costa Souza de Araujo

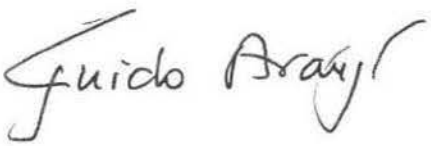
Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Compressão de dados (Computação). 2. Compiladores. 3. Linguagens de Programação (Computadores). I. Araujo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Compressão de Byte-Code Utilizando Recuperação de Sintaxe Abstrata

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Bruno Kraychete da Costa e aprovada pela Banca Examinadora.

Campinas, 22 de março de 2000.



Prof. Dr. Guido Costa Souza de Araújo
(IC-UNICAMP) (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 22 de março de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Edna Natividade da Silva Barros
UFPE

Prof. Dr. Luiz Eduardo Buzato
IC-UNICAMP

Prof. Dr. Guido Costa de Souza Araújo
IC-UNICAMP

Resumo

Com os avanços da tecnologia nas áreas de comunicação sem fio e computação pessoal, os chamados *computadores portáteis* ou *handhelds* estão se tornando alvo de muita atenção em Computação. Por outro lado, linguagens cujo alvo são máquinas virtuais (ex. *Java*), têm permitido a construção de sistemas portáteis distribuídos que são independentes do processador. Os *handhelds* são normalmente projetados em torno de um processador dedicado e uma pequena quantidade de memória, de modo a manter baixo o custo do projeto. As limitações de memória encontradas em tais dispositivos, e o aumento da demanda por banda disponível nas redes têm renovado o interesse em pesquisas voltadas para a compressão de código móvel. Neste trabalho, nós mostramos um método de compressão para estes dispositivos baseado na recuperação das Árvores de Sintaxe Abstrata (ASA) do programa. Resultados experimentais, utilizando o Benchmark JVM98, revelaram uma melhoria nas razões de compressão em relação àquelas obtidas usando exclusivamente o algoritmo Lempel-Ziv-Welch.

Abstract

With the advances in technologies for wireless communications and personal computing, the so called *portable appliances* or *handhelds* are expected to become mainstream computer devices. On the software side, languages targeted to virtual machines (eg. *Java*) have enabled the design of flexible mobile distributed systems that are independent of the underlying processor. In order to keep costs low, handhelds are usually designed around a cheap dedicated processor and a small memory set. The memory size constraints found on such devices, and the increasing demand for bandwidth have been renewing the research interest towards new program compression techniques. In this work, we show a compression method for these devices, that is based on the recovery of the program Abstract Syntax Trees. Experimental results, using the JVM98 Benchmark, reveal an improvement on the compression ratio over the LZW algorithm when the ASTs are recovered before compression.

Agradecimentos

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico — CNPq (proc. 131.947/98-2) pela confiança e pelo apoio financeiro.

Ao meu orientador, Guido Araújo, pela excelente orientação apesar de toda a sobrecarga, pelo convívio agradável e pelas grandes experiências de vida e ensinamentos transmitidos.

Aos meus pais e toda minha família pelo incentivo e apoio incondicionais. Este trabalho é dedicado a vocês.

À minha noiva, Diala, pelo amor, carinho, compreensão, incentivo e apoio que nunca me faltaram.

Aos amigos Ivan, Leonardo, Marcelo, Nathan, Paulo e Sandro pela amizade e companheirismo nas noites de trabalho nos laboratórios do Instituto de Computação.

A todos os amigos da *Cleide's House* pela convivência e pela experiência de vida.

Aos funcionários e colegas do Instituto de Computação pelo ótimo ambiente de trabalho.

À Deus e a meu Anjo da Guarda, pela luz, força de vontade e coragem de seguir adiante e por nunca terem me desamparado. Por me permitirem a conclusão deste trabalho e por terem colocado em meu caminho pessoas tão agradáveis de se conviver.

BRUNO K. DA COSTA

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A demanda crescente por comunicação e dispositivos móveis (*handhelds*) deu início a mudanças profundas na forma como os computadores são projetados e utilizados. Nos últimos meses, temos testemunhado a introdução de novas tecnologias para *handhelds*. Algumas delas tornarão possíveis a transmissão de dados sem a necessidade de fios (ex. Blueto-oth [17]) ou a conexão de dispositivos na rede local sem a necessidade de configurações prévias (ex. Jini [20]). É esperado que em cinco anos o mercado de sistemas portáteis (ou *embedded*) irá ultrapassar o mercado de computadores *desktop* [9], um crescimento anual de cerca de 40%, maior que os 12% esperados para o mercado de *desktops*. Este fato tem motivado uma forte mudança da pesquisa na indústria e nas universidades na direção do desenvolvimento de computadores portáteis. Estes dispositivos têm que obedecer fortes limitações tais como custo e tamanho reduzidos e baixa dissipação de energia, pois são projetados para mercados muito competitivos. Por este motivo, os *handhelds* são normalmente construídos utilizando-se processadores dedicados de baixo custo e pequenas quantidades de memória. Com o aumento do tamanho das aplicações [18], compressão de programas (código) tem se tornado um importante problema nesta área. Além disso, a compressão de código pode também diminuir a quantidade de dados transferidos entre estes sistemas, melhorando consideravelmente o uso da banda disponível.

1.1 Trabalhos Correlatos

Um grande número de técnicas tem sido propostas na literatura para o problema da compressão de código. As mais utilizadas normalmente são baseadas no algoritmo de Lempel-Ziv [22] e suas variações [5]. Ernst et al [11] propõem uma representação intermediária comprimida e executável, chamada BRISC (*Byte-Code RISC*), baseada na captura de árvores de expressão do programa, uma idéia originalmente desenvolvida por Proebsting [24] utilizando o conceito de *superoperators*. O resultado desta abordagem são

razões de compressão[1] próximas das resultantes do gzip [25] quando utilizado na plataforma x86. A abordagem utilizada na BRISC comprime código nativo das plataformas x86, SPARC e PowerPC. É importante ressaltar que aqui menor razão de compressão significa maior capacidade de compressão, gerando arquivos comprimidos menores. Neste mesmo trabalho, os autores também propuseram outra representação, chamada *wire code*, que resulta em razões de compressão cerca de 40% melhores que o gzip. Este resultado foi obtido através da combinação da melhor seqüência de técnicas de compressão e codificação que, quando combinadas, resultam em razões de até 20,4%. Esta técnica foi utilizada para comprimir código nativo da plataforma SPARC. Devido ao fato das duas abordagens de compressão descritas no trabalho de Ernst et al utilizarem como alvo o código nativo de várias plataformas, não foi possível a comparação direta desses métodos com as abordagens apresentadas neste trabalho.

Em [15], é descrito um compilador C que é capaz de gerar código comprimido executável e um interpretador para o mesmo. A Microsoft Research está atualmente trabalhando no Projeto Sumatra, uma abordagem para compressão de código baseado no trabalho de Fraser [11], que deve ser introduzido em máquinas virtuais para *handhelds*. Em [7] a alocação de registradores é utilizada para transformar seqüências de instruções diferentes em seqüências similares. Debray et al [8] e Baker et al [4] usam informações de (pós)dominância [1] para identificar seqüências de blocos básicos similares que são então codificados utilizando técnicas de abstração procedural.

Franz et al [12, 13] propuseram uma abordagem de Árvores de Sintaxe Abstrata (ASA) adaptativas para código-objeto. Uma abordagem similar para seqüência de byte-codes foi relatada em [14], enquanto este trabalho estava em desenvolvimento. Neste último trabalho, os autores propõem uma alternativa ao uso do byte-code como código compilado Java. Eles passam a utilizar uma estrutura baseada em árvores que é duas vezes mais compacta que o equivalente em byte-code. Mais informações de alto nível estão contidas nesta nova representação, permitindo que sejam realizadas otimizações antes impossíveis de serem realizadas. Além disso, os autores desenvolveram um protótipo de Máquina Virtual Java que não só é capaz de executar código na nova representação, como também é capaz de realizar otimização em *real-time*. Enquanto o código é executado, informações de *profiling* são obtidas e alimentam um otimizador que por sua vez devolve o código otimizado para a unidade de execução. É importante ressaltar que, apesar deste trabalho apresentar bons resultados, ele implementa profundas mudanças na estrutura atual do código Java, enquanto que neste trabalho nós propomos abordagens para compressão partindo do próprio byte-code.

Com o aumento da popularidade de dispositivos portáteis e redes desenvolvidas para operar com tais dispositivos, esperamos um aumento nas pesquisas voltadas para repre-

[1] *Razão de Compressão = Tamanho do programa comprimido / Tamanho do programa original × 100 (%)*

sentações comprimidas de programas.

1.2 Metodologia de Trabalho

O objetivo deste trabalho era conseguir desenvolver um método de compressão para código Java baseado nos trabalhos de Franz et al [13] e Araujo et al [3]. O que observamos é que, apesar dos métodos descritos nestes trabalhos apresentarem bons resultados em código nativo SPARC e MIPS, o mesmo não ocorreu quando os mesmos métodos foram aplicados em código Java. O que normalmente acontece é que para códigos nativo, o compilador gera vários módulos (código objeto) que são ligados no final do processo num único executável. Não acontece o mesmo com Java. Programas em Java normalmente são semanticamente separados em classes que quando compiladas geram vários arquivos de classe Java com tamanho bastante reduzido quando comparado a módulos em código nativo. Esses arquivos de classe (ou módulos) não são ligadas num único executável. Um deles é invocado e os outros são carregados automaticamente e sob demanda pela Máquina Virtual Java. Assim, para programas similares, é comum que o tamanho da versão para código nativo supere o tamanho da versão em Java em muitas vezes. Os algoritmos de compressão têm condições de encontrar maior redundância em porções maiores de código e por isso, apresentam desempenho inferior quando aplicados em pequenos trechos. Este problema poderia ser resolvido através do agrupamento de todas as classes pertencentes a um mesmo projeto num único arquivo, chamado *archive Java* ou mais comumente *jar file*. Isso já é feito atualmente pelas implementações de Java utilizando o algoritmo LZW. Nossa proposta seria utilizar a mesma abordagem substituindo o LZW pelos métodos de compressão propostos neste trabalho. Contudo, devido a problemas de tempo, não fomos capazes de implementar tal recurso. Também por restrições de tempo, a descompressão dos métodos aqui propostos não foi implementada. Porém, a real contribuição deste trabalho se apresenta na forma de métodos de fatoração e detecção de expressões comuns, bem como técnicas de decompilação de byte-code.

Nosso trabalho está disposto da seguinte forma. O capítulo 2 apresenta uma breve descrição da Máquina Virtual Java e como se dá seu funcionamento. Apresenta também a estrutura de um programa Java compilado (arquivo de classe Java) e mostra alguns exemplos de programas Java e seus equivalentes em byte-code. O capítulo 3 mostra algumas tentativas realizadas para se conseguir boas razões de compressão. Os métodos de casamento de padrões e casamento de prefixos/sufixos são explicados e alguns resultados são mostrados. O nosso método principal de compressão, baseado na recuperação da ASA está descrito no capítulo 4, onde também são mostrados resultados experimentais. Por fim, no capítulo 5, nós concluímos nosso trabalho e indicamos possíveis abordagens para trabalhos futuros.

Capítulo 2

A Máquina Virtual Java

Neste capítulo descrevemos sucintamente o funcionamento e a organização da Máquina Virtual Java (JVM). Apresentamos também exemplos de programas Java e seu correspondentes compilados. As informações presentes neste capítulo serão utilizadas ao longo dos capítulos seguintes. Maiores detalhes sobre o funcionamento da JVM podem ser encontrados em [23].

2.1 Descrição

A linguagem Java foi introduzida em 1995 pela Sun Microsystems. A grande motivação por trás da criação de Java foi a necessidade de se projetar uma linguagem extremamente portável que pudesse ser executada em redes de sistemas móveis baseados em plataformas heterogêneas. O alvo neste caso eram dispositivos conhecidos como *network applicances*. A idéia fundamental por trás deste paradigma, era o de se criar um único compilador capaz de traduzir programas escritos em Java para uma mesma representação intermediária, ao invés de criar compiladores diferentes para cada plataforma. Os programas escritos em Java são compilados para uma representação intermediária não executável chamada *byte-code*. Byte-code é uma espécie de código de máquina que só pode ser executado em uma máquina virtual denominada *Java Virtual Machine* (JVM). A abordagem que a Sun escolheu envolve criar versões desses ambientes de execução para diversas plataformas. Durante a execução de um programa Java ocorrem então as seguintes etapas: primeiro, traduz-se o programa para byte-code, utilizando um compilador Java e em seguida submete-se o programa à máquina virtual que o interpretará diretamente. Como resultado deste processo, costuma-se dizer que os programas em Java são compilados e interpretados. A seguir descrevemos algumas das características da JVM que serão úteis no desenrolar deste trabalho.

2.1.1 Arquitetura da JVM

A JVM é composta dos seguintes elementos básicos:

- PC - Contador de programa. Cada *thread* de execução possui um contador de programa que mantém o endereço da próxima instrução a ser executada.
- Pilha de Operandos. Cada *thread* possui uma pilha que, tal como uma pilha em C, serve para armazenar variáveis locais, valores parciais e endereço de retorno.
- *Heap*. Compartilhada entre todas as *threads* em execução, a *heap* é a área de memória onde serão alocados todos os objetos criados dinamicamente.
- Área de Métodos. Similar ao *"text" segment* em C, é nesta área, também compartilhada entre todas as *threads*, que se encontram as estruturas de classes (tabela de constante, dados de atributos e métodos — inclusive construtores e métodos especiais).
- Tabela de Constantes. Se assemelha à tabela de símbolos do C, diferenciando-se no fato de armazenar mais informações. Cada classe ou interface possui uma Tabela de Constantes e nela estão contidas informações sobre as constantes definidas, referências a métodos e atributos que serão resolvidos em tempo de execução.
- Pilha para Métodos Nativos. Esta pilha é utilizada para simular o conjunto de instruções da máquina virtual numa linguagem como C. Também pode ser utilizada por métodos em Java, dependendo apenas da implementação da máquina virtual.

2.1.2 Funcionamento Básico

Assim como em C, registros de ativação (ie. *frames*) são utilizados na JVM também para armazenamento de dados e resultados parciais, bem como para ligação dinâmica, para retornar valores de métodos e disparar exceções. Todo *frame* é alocado na pilha Java da *thread* que o criou, e é criado no momento em que um método é invocado e destruído quando o método é encerrado, tenha o encerramento sido normal ou anormal (ex.: ocorrência de exceções). É importante notar que cada *frame* é local à *thread* e não pode ser referenciado externamente. Um *frame* é implementado como um *array* de *words* onde cada variável é acessada pelo índice que ocupa no *array*. Variáveis dos tipo *long* e *double* ocupam duas entradas no *array* e são referenciadas pelo índice menor.

Pilha de Operandos

A Pilha de Operandos é alocada no *frame* a cada invocação de um método, e serve para armazenar os argumentos das instruções da JVM. A JVM é uma máquina de pilha e, por isso, necessita de uma pilha para armazenar temporariamente os resultados das computações. Para uma adição, por exemplo, é necessário carregar na pilha os dois valores a serem computados e, em seguida, invocar a instrução de soma, que irá remover os dois itens da pilha e empilhar o resultado da operação. Uma parte da JVM, denominada *Class File Verifier*, impede que se faça mal uso da Pilha de Operandos (ex.: tratar duas entradas do tipo *int* como uma entrada do tipo *long*).

Ligação Dinâmica

Um *frame* da JVM possui uma referência na Tabela de Constantes para o tipo do método corrente, permitindo assim o carregamento dinâmico do código deste. O *Arquivo de Classes* (a ser descrito na seção 2.2) possui referências simbólicas às variáveis e métodos. Durante a ligação dinâmica, referências simbólicas são resolvidas e transformadas em referências concretas, classes adicionais são carregadas sob demanda e os acessos às variáveis do método são traduzidas para *offsets* dentro das estruturas de armazenamento usadas em tempo de execução.

Encerramento de Métodos

Os métodos podem ser encerrados normal ou anormalmente. Uma invocação de método termina normalmente caso a invocação não cause uma exceção. Se a invocação termina normalmente, um valor será retornado para seu invocador. Neste caso, o *frame* da JVM é utilizado para restaurar o estado do invocador, incluindo suas variáveis locais e o estado de sua Pilha de Operandos, e para ajustar apropriadamente o Contador de Programa do método invocador.

Uma invocação de método termina anormalmente se a execução de uma instrução da JVM causa uma exceção e esta exceção não é tratada dentro do método. A avaliação da instrução `throw` também causa o disparo de uma exceção. Um método que termina anormalmente nunca retorna um valor para seu invocador. Neste caso, a JVM realiza alguns ajustes no *frame* para que a execução continue no método invocador.

Tratamento de Exceções

Em geral, o disparo de um exceção resulta na imediata transferência do controle, que pode saltar múltiplas sentenças ou invocações de construtores e/ou métodos em Java, até

encontrar a cláusula *catch*. Se a cláusula *catch* não for encontrada, e neste caso dizemos que o método corrente tem uma "exceção não tratada", o método corrente termina anormalmente. Sua Pilha de Operandos e variáveis locais são descartadas, seu *frame* é removido da pilha Java, e o *frame* do método invocador reinstalado. A exceção é então re-disparada no contexto do método invocador e assim por diante até que uma de duas situações ocorra: é encontrada uma cláusula *catch* apropriada para a exceção disparada, e neste caso a exceção é tratada e o fluxo de execução continua normalmente; ou nenhuma cláusula é encontrada e o topo da cadeia de invocação de métodos é alcançado. Neste último caso, o fluxo de execução da *thread* é interrompido e uma mensagem de "exceção não tratada" é informada.

Informações Adicionais

Um *frame* de uma JVM pode ser estendido para armazenar outros tipos de informação, tais como informações de depuração. Estas extensões estão previstas na Especificação da Máquina Virtual Java [23], mas são dependentes da implementação.

2.1.3 Conjunto de Instruções

Uma instrução da JVM consiste de um *operador*, um byte que especifica a ação a ser realizada, seguido de zero ou mais *operandos* que são os argumentos ou dados que serão utilizados na realização da operação. O número e o tamanho dos operandos é determinado pelo operador em questão. Se um operando ocupa mais de um byte, então ele é representado no formato *big-endian*, onde o byte mais significativo vem primeiro. Por exemplo, um índice de 16 bits para um variável é representado por dois bytes, *byte1* e *byte2*. O cálculo do índice é então dado por $(byte1 << 8 \mid byte2)$. A decisão de restringir o tamanho do operadores da JVM em um byte mostra a preocupação de seus projetistas em limitar o crescimento dos programas em byte-code. Isto é facilmente compreendido, tendo em mente que a JVM foi originalmente projetada para executar programas em sistemas portáteis com muito pouca memória.

Muitos operadores da JVM codificam o tipo de seus argumentos na própria instrução. Por exemplo, o operador *iload* carrega na Pilha de Operandos o conteúdo de uma variável, que deve ser inteira. O operador *fload* faz a mesma operação com uma variável *float*. A implementação de ambos os operadores é indêntica, mas cada um possui um opcode diferente. Na maioria dos casos, o tipo associado ao operador é distinguido através de uma única letra: *i* para operações com inteiros, *l* para inteiros longos, *s* para inteiros curtos, *b* para bytes, *c* para caracteres (char), *f* para ponto flutuante de precisão simples, *d* para ponto flutuante de precisão dupla e *a* para referências. Algumas instruções, porém, não apresentam ambigüidade no uso e, por isso, não se utilizam desses mnemônicos especia-

lizados. Por exemplo, o operador *arraylength* sempre opera numa referência a um *array* e retorna o seu tamanho. Operadores como *goto* e operadores de desvio condicional não usam operandos tipados.

2.1.4 Tipos de Dados

A JVM aceita diretamente, em byte-code, alguns dos tipos utilizados na linguagem Java. Entre eles estão os tipos integrais tais como *byte*, *short*, *int*, *long* e *char*; e tipos de ponto flutuante como *float* e *double*. A JVM também tem suporte explícito a objetos, ou seja, é possível armazenar referências a objetos em variáveis, sejam eles *arrays* ou instâncias dinamicamente alocadas de classes Java. Esse tipo de dado é chamado de *reference*. Além dos tipos convencionais presentes em várias linguagens de alto nível, a JVM suporta também um tipo especial que não possui nenhum correspondente direto em Java, o *returnAddress*, usado para guardar endereços de retorno. Todos esses tipos podem ser inferidos pelo tipo de instrução que se usa para referenciar um determinado dado. Veremos alguns exemplos na seção 2.3.

A tabela 2.1 resume o suporte a tipos no conjunto de instruções da JVM. Apenas as instruções que existem para mais de um tipo estão listadas. Uma instrução específica, com a informação do tipo, pode ser construída substituindo-se o T nas instruções na coluna opcode pela letra na coluna do tipo. Se a coluna para um determinado tipo está em branco, significa que não há uma instrução para aquele tipo de operando. Por exemplo, existe uma instrução de armazenamento (*store*) para o tipo inteiro, *istore*, mas não existe uma instrução *store* para o tipo *byte*.

Note que a maioria das instruções na tabela 2.1 não tem uma forma para os tipos integrais *byte*, *char* e *short*. Quando a JVM precisa escrever em variáveis destes tipos ou manipular valores destes tipos na Pilha de Operandos, ela faz a extensão com sinal dos tipos *byte* e *short* para *int* e faz a extensão com zeros de *char* para *int*. Desta forma, a maioria das operações para os tipos *byte*, *char* ou *short* são corretamente realizadas em variáveis do tipo *int*.

A associação entre o tipos de armazenamento do Java e os tipos computacionais em byte-code (tipos efetivamente utilizados do ponto de vista do espaço) estão sumarizados na tabela 2.2.

Arrays dos tipos *boolean*, *byte*, *char* e *short* podem ser diretamente representados pela JVM. *Arrays* dos tipos *byte*, *char* e *short* são acessados utilizando-se operadores apropriados para estes tipos, enquanto *arrays* do tipo *boolean* utilizam as instruções do tipo *byte*.

A seguir descrevemos o funcionamento de algumas das instruções mais importantes da JVM, para assim melhor entender os exemplos descritos na seção 2.3.

opcode	byte	short	int	long	float	double	char	reference
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2f*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Tabela 2.1: Suporte a tipos no conjunto de instruções da JVM

Tipo em Java	Tamanho em bits	Tipo Computacional
byte	8	int
char	16	int
short	16	int
int	32	int
long	64	long
float	32	float
double	64	double

Tabela 2.2: Tipos na Máquina Virtual Java e seus tamanhos em bits

As instruções que operam com a Pilha de Operandos são as seguintes:

- Carregam o valor de uma variável local na Pilha de Operandos:
  iload, *iload_<n>*, *lload*, *lload_<n>*, *fload*, *fload_<n>*, *dload*, *dload_<n>*, *aload*, *aload_<n>*.
- Armazenam o valor no topo da Pilha de Operandos numa variável local:
  istore, *istore_<n>*, *lstore*, *lstore_<n>*, *fstore*, *fstore_<n>*, *dstore*, *dstore_<n>*, *astore*, *astore_<n>*.
- Carregam uma constante na Pilha de Operandos:
  bipush, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*, *lconst_<l>*, *fconst_<f>*, *dconst_<d>*.
- Permite acesso a mais variáveis locais utilizando um índice maior (de dois bytes), ou a um operando imediato maior:
  wide.

Instruções para acesso a atributos de objetos ou elementos de um *array* também copiam valores de e para a Pilha de Operandos.

Os mnemônicos mostrados acima, terminados com uma letra entre os sinais < e > (ex.: *iload_<i>*), denotam uma família de instruções. Tais famílias de instruções são especializações de uma instrução adicional genérica (ex.: *load*) que recebe um operando. Para as instruções especializadas, alguns operandos são implícitos e não necessitam ser armazenados ou lidos do programa objeto. No caso de *iload_<i>*, o operando i é codificado diretamente na instrução.

As instruções aritméticas estão organizadas de forma similar às instruções de carregamento e salvamento de variáveis. Elas são utilizadas para computar resultados típicos de

operações que obtêm dois operandos da Pilha de Operandos e lá colocam o resultado da operação. As instruções aritméticas são apresentadas a seguir:

- Adição: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtração: *isub*, *lsub*, *fsub*, *dsub*.
- Multiplicação: *imul*, *lmul*, *fmul*, *dmul*.
- Divisão: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Resto de Divisão: *irem*, *lrem*, *frem*, *drem*.
- Negação: *ineg*, *lneg*, *fneg*, *dneg*.
- Deslocamento: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- OU Lógico: *ior*, *lor*.
- E Lógico: *iand*, *land*.
- OU Exclusivo Lógico: *ixor*, *lxor*.
- Incremento para Variável Local: *iinc*.

Há também na JVM as instruções de conversão de tipo. Elas têm o formato *a2b*, e transformam valores na Pilha de Operandos do tipo *a* para o tipo *b*.

Como já foi dito, a JVM suporta o conceito de referência e conta com instruções especiais para criação e manipulação de objetos. Apesar do conceito de objetos na JVM englobar instâncias de classes e *arrays*, há conjuntos distintos de instruções para acesso a cada uma dessas abstrações.

- Criação de nova instância de classe:
  new.
- Criação de um novo *array*:
  newarray, *anewarray*, *multianewarray*.
- Acesso a atributos de classes (estáticos e não estáticos):
  getfield, *getstatic*, *putfield*, *putstatic*.
- Carregamento de um elemento de um *array*:
  baload. *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.

- Armazenamento no elemento de um *array*:
  bastore. castore, sastore, iastore, lastore, fastore, dastore, aastore.

- Tamanho de um *array*:
  arraylength.

- Checar propriedades de uma instância de classe ou array:
  instanceof, checkcast.

O tratamento de desvios na JVM é feito por intermédio de famílias de instruções especializadas:

- Desvio condicional:
  ifeq, ifne, iflt, ifle, ifgl, ifge, ifnull, if_icmpeq, if_icmpne, if_icmplt, if_icmple, if_icmpgt, if_icmpge, if_acmpeq, if_acmpne, lcmp, fcmpl, fcmpg, dcmpl, dcmpg.

- Desvio condicional composto:
  tableswitch, lookupswitch.

- Desvio incondicional:
  goto, goto_w, jsr, jsr_w, ret.

Para invocação de métodos existem quatro instruções:

- Invocação de método de instância (invocação usual):
  invokevirtual.

- Invocação de método de especiais (construtor, método privado ou métodos da superclasse):
  invokespecial.

- Invocação de método implementado numa interface:
  invokeinterface.

- Invocação de método de classe (método estático):
  invokestatic.

Por fim, existem as instruções de disparo de exceção (*athrow*), instruções para implementação da palavra-chave Java *finally* usada em tratamento de exceções (*jsr*, *jsr_w* e *ret*), e as instruções de sincronização para execução concorrente (*monitorenter* e *monitorexit*).

2.2 Estrutura de um Arquivo de Classe

Nesta seção trataremos do formato do arquivo de classes Java. Esta informação é importante pois com isto é possível entender como a Máquina Virtual carrega classes e como se utiliza da Tabela de Constantes para referenciar constantes e métodos.

Cada arquivo de classe Java contém a definição de apenas uma classe ou interface. Um arquivo de classe consiste de uma seqüência de bytes. Todo e qualquer dado de 16, 32 ou 64 bits é construído lendo-se 2, 4 ou 8 bytes consecutivos, respectivamente. Itens formados por mais de um byte estão sempre armazenados em ordem *big-endian*, onde o byte mais significativo vem primeiro.

Para melhor explicar o formato do arquivo de classes Java, definiremos alguns tipos de dados. Os tipos u2, u4 e u8 representam, respectivamente, valores sem sinal formados por 2, 4 ou 8 bytes. O formato do arquivo de classe é apresentado utililizando-se pseudo-estruturas em notação semelhante a linguagem C. Tabelas de tamanho variável são usadas em várias estruturas do arquivo de classes e, embora nós utilizemos a sintaxe de *arrays* em C, não é possível endereçar diretamente um item da tabela.

A estrutura de um arquivo de classes Java tem o formato apresentado na figura 2.1.

```
ClassFile {
    u4 magic;
    u2 minorVersion;
    u2 majorVersion;
    u2 constantPoolCount;
    cp_info constantPool[constantPoolCount-1];
    u2 accessFlags;
    u2 thisClass;
    u2 superClass;
    u2 interfacesCount;
    u2 interfaces[interfacesCount];
    u2 fieldsCount;
    field_info fields[fieldsCount];
    u2 methodsCount;
    method_info methods[methodsCount];
    u2 attributesCount;
    attribute_info attributes[attributesCount];
}
```

Figura 2.1: Estrutura de um arquivo de classe Java

A descrição de cada item da estrutura *ClassFile* segue abaixo:

magic

É a assinatura de um arquivo de classe. Seu tipo é um *long* e sempre tem valor igual a 0xCAFEBABE.

minorVersion e majorVersion

Refletem a versão do compilador que gerou o arquivo de classe. Seus valores servem para que o interpretador verifique se há condições de se executar o código da classe. Para o *Java Developer's Kit* (JDK) 1.0.2 da Sun, os valores documentados são 45 para *majorVersion* e 3 para *minorVersion*. Esses valores são os mesmos para o JDK 1.1.x.

constantPoolCount

Este valor representa o número de entradas na Tabela de Constantes (constantPool) do arquivo de classe. Este valor deve ser maior que zero. Diferentemente de *arrays* em C, o primeiro elemento desta tabela tem índice 1 e um índice é considerado válido se for maior que zero e menor que o valor do que *constantPoolCount*.

constantPool

É a Tabela de Constantes propriamente dita. Possui tamanho variável e é composta de outras estruturas também de tamanho variável representando strings, nomes de classes e atributos, e outras constantes. A primeira entrada na *constantPool*, *constantPool[0]*, é reservada para uso interno da implementação da JVM. Esta entrada não está presente no arquivo de classe. A primeira entrada presente no arquivo de classe é a entrada *constantPool[1]*. Cada entrada entre os índices 1 e *constantPoolCount - 1* é uma estrutura de tamanho variável cujo formato é determinado pelo primeiro byte, chamado *tag*.

accessFlag

O valor de *accessFlag* é uma máscara de modificadores utilizada na declaração de classes e interfaces. Os valores possíveis para *accessFlags* são apresentados na tabela 2.3.

ACC_INTERFACE é utilizado para diferenciar uma classe de uma interface. Uma interface é implicitamente abstrata; a *flag* ACC_ABSTRACT deve estar setada. Uma interface não pode ser final (ACC_FINAL deve ficar ressetada), de outra forma sua implementação não seria possível.

As *flags* ACC_FINAL e ACC_ABSTRACT não podem estar setadas simultaneamente. A *flag* ACC_SUPER em classes geradas por compiladores mais recentes está sempre setada. Esta *flag*, quando setada, determina que os métodos da superclasse devem ser invocados através do operador *invokespecial*.

Nome da Flag	Valor	Significado	Usado por
ACC_PUBLIC	0x0001	Acesso público	Classe, interface
ACC_FINAL	0x0010	Final. Não são permitidas subclasses	Classe
ACC_SUPER	0x0020	Obsoleto	Classe, interface
ACC_INTERFACE	0x0200	É uma interface	Interface
ACC_ABSTRACT	0x0400	Abstrata. Não pode ser instanciada	Classe, interface

Tabela 2.3: Possíveis valores para *accessFlag*

thisClass

Este atributo deve ser um índice válido da Tabela de Constantes. A entrada da Tabela de Constantes indexado por este valor deve ser uma estrutura que representa a classe ou interface definida neste arquivo de classe.

superClass

superClass deve ser zero para classes (indicando que a classe em questão é derivada diretamente de java.lang.Object) ou um índice válido. Se este valor não for zero, ele deve apontar para uma entrada na Tabela de Constantes representando a superclasse da qual esta classe derivou. Para uma interface, o valor de *superClass* deve ser sempre um índice válido para a Tabela de Constantes.

interfacesCount

Determina a quantidade de superinterfaces diretas desta classe ou interface.

interfaces[]

Cada entrada deste *array* deve ser um índice válido para a Tabela de Constantes e deve apontar para uma estrutura representando a interface que é uma superinterface direta da classe ou interface em questão.

fieldsCount

Número de atributos da classe ou interface em questão. Determina o número de entradas na estrutura *fields* descrita a seguir.

fields[]

Cada entrada neste *array* deve ser uma estrutura de tamanho variável dando uma descrição completa de um atributo da classe ou interface. A tabela *field*[] inclui somente aqueles atributos que foram declarados nesta classe ou interface. Ela não inclui os atributos declarados e herdados de superclasses ou superinterfaces.

methodsCount

Número de entradas presentes na estrutura *methods*[].

methods[]

Cada entrada no *array methods*[] é um estrutura de tamanho variável dando a descrição completa de um método da classe ou interface em questão. As estruturas contidas em *methods*[] representam todos os métodos, sejam eles de instância ou de classe, declarados nesta classe ou interface. O *array methods*[] não inclui nenhum método declarado em superclasses ou superinterfaces.

attributesCount

Determina o número de atributos no *array attributes*[] definido a seguir.

attributes[]

Cada entrada deste *array* deve ser uma estrutura de tamanho variável que representa um atributo da classe. Um arquivo de classe pode ter qualquer número de atributos associados. O único atributo definido na especificação para o *array attributes*[] é o *SourceFile*, que representa o nome do arquivo que, quando compilado, deu origem a classe em questão.

2.3 Exemplos de Programas em Byte-code

Nesta seção apresentamos e analisamos alguns trechos de código Java com suas respectivas traduções em byte-code. Propositalmente omitiremos as informações presentes nas respectivas Tabelas de Constantes. Desta forma, quando apresentamos um operador da JVM que usa um índice para a Tabela de Constantes como operando, a exemplo de: *invokevirtual 13* estamos assumindo que existe uma entrada na Tabela de Constantes, cujo índice (13 no exemplo dado) aponta para a estrutura correta.

Inicialmente, vejamos alguns exemplos básico de atribuições (figura 2.2). Mostramos na figura 2.2 seis atribuições. As três da direita são atribuições para referências, enquanto

que as da esquerda, atribuições para variáveis com tipos numéricos. Os primeiros quatro exemplos mostram que o código em byte-code gerado carrega um valor (uma constante, no caso) na pilha e em seguida o associa a uma variável local da JVM. Os últimos dois exemplos mostram a atribuição do valor de um variável para outra variável local. O acesso a atributos de objetos é feito de forma ligeiramente diferente. Enquanto as variáveis locais podem ser acessadas diretamente através de um número, os atributos necessitam da referência ao objeto. Veja, por exemplo, a figura 2.3.

Java	Byte-Code	Java	Byte-Code
int a = 0;	iconst_0 istore_0	String s = "teste";	ldc 3 astore_1
double d = 1.1;	ldc2_w 12 dstore_2	Object o = null;	aconst_null astore 4
float f1 = f2;	fload 5 fstore 6	String str = s;	aload_1 astore 7

Figura 2.2: Exemplos de atribuição

Escrita em Atributo

Java	Byte-Code
car.engine = null;	aload_0 const_null putfield 25

Leitura de Atributo

Java	Byte-Code
Engine e = car.engine;	aload_0 getfield 25 astore_1

Figura 2.3: Exemplos de acesso a atributos de um objeto

Suponha que desejamos acessar o atributo *engine* do objeto *car*. Do lado esquerdo da figura 2.3 nós atribuímos um valor a tal atributo, enquanto que no lado direito nós realizamos a leitura do mesmo. Para realizar a escrita de um atributo primeiramente empilhamos a referência ao objeto (aload_0), depois carregamos a referência nula, para então atribuir o valor ao atributo *engine*, representado pelo índice 25 (putfield 25) da Tabela de Constantes. No caso da leitura, os passos são bastante semelhantes. Primeiramente, a referência ao objeto cujo atributos gostaríamos de ler é empilhada. O atributo a acessar é então selecionado (getfield 25), após o qual podemos fazer uso de seu valor a partir do topo da pilha. Neste caso armazenamos o atributo em uma variável local (variável *e* do tipo *Engine*) através da instrução astore_1.

Um exemplo de alocação de um objeto pode ser visto na figura 2.4. Ali, a alocação de uma string é traduzida em cinco instruções em byte-code. A primeira realiza a alocação do objeto descrito pela entrada da Tabela de Constantes cujo índice é 88. A segunda instrução duplica a referência ao objeto para servir de parâmetro ao construtor, invocado mais adiante. A terceira instrução (ldc 37) empilha o argumento ao construtor (constante string "Ok") enquanto a quarta, finalmente invoca o construtor consumindo da pilha a referência ao objeto. O resultado, a referência ao objeto já inicializado, é por fim atribuído a uma variável local através da instrução astore_4.

Java	Byte-Code
String s = new String("Ok");	new 88 dup ldc 37 invokespecial 76 astore_4

Figura 2.4: Alocação de objetos em byte-code

O próximo exemplo mostra como é traduzido um bloco de decisão (`if-then-else`). A figura 2.5 mostra como esta instrução de alto nível é convertida em uma seqüência de byte-codes.

O teste da instrução `if` em Java é traduzido em dois operadores, o `iload_0` e o `ifeq L1` que carrega o valor de *flag* na pilha e desvia para o rótulo L1 caso seja zero (falso). Caso este valor seja diferente de zero, as instruções seguintes são executadas. Elas apenas imprimem uma mensagem de "Ok" na saída padrão.

Java	Byte-Code

`if (flag) {`
`  System.out.println(''Ok'');`
`}`
`...`
	`  iload_0`
	`  ifeq L1`
	`  getstatic 10`
	`  lcd 12`
	`  invokevirtual 19`
	`L1: ...`

Figura 2.5: Exemplo de bloco de decisão em byte-code

Na figura 2.6 nós podemos verificar um exemplo de laços dos tipos WHILE e DO-WHILE. Os exemplos são auto-explicativos. Note que, do ponto de vista do código em

byte-code, a única coisa que muda do laço WHILE para o laço DO-WHILE é a existência do operador *goto L2* no primeiro. Este desvio incondicional permite saltar diretamente para o teste do laço antes de qualquer interação do mesmo. É importante salientar que a forma de construção do bloco WHILE aqui apresentada corresponde àquela gerada pelo compilador do *Java Developer's Kit* da Sun. Outros compiladores podem gerar seqüências diferentes de operadores em byte-code. Uma forma alternativa de apresentação de laço WHILE é colocar o teste do laço no início do bloco e não no final como o exemplo apresentado.

Exemplo de WHILE

Java	Byte-Code

`i = 0;`
`while(i < 10) {`
`  System.out.println(i);`
`  i++;`
`}`
`...`
	`  iconst_0  ; carrega 0 na pilha`
	`  istore_1  ; atribui topo da pilha`
	`  a variável 1 (i)`
	`  goto L2  ; salta para o teste`
	`L1: getstatic 10  ; carrega obj System.out`
	`  iload_1  ; carrega o valor de i`
	`  invokevirtual 19 ; System.out.println(i)`
	`  iinc 1, 1  ; incrementa i`
	`L2: iload_1  ; carrega i`
	`  bipush 10  ; carrega o byte 10`
	`  ifcmplt L1  ; salta p/ L1 se i < 10`
	`  ...`

Exemplo de DO-WHILE

Java	Byte-Code

`i = 0;`
`do {`
`  System.out.println(i);`
`  i++;`
`} while(i < 10);`
`...`
	`  iconst_0  ; carrega 0 na pilha`
	`  istore_1  ; atribui topo da pilha`
	`  a variável 1 (i)`
	`L1: getstatic 10  ; carrega obj System.out`
	`  iload_1  ; carrega o valor de i`
	`  invokevirtual 19 ; System.out.println(i)`
	`  iinc 1, 1  ; incrementa i`
	`L2: iload_1  ; carrega i`
	`  bipush 10  ; carrega o byte 10`
	`  ifcmplt L1  ; salta p/ L1 se i < 10`
	`  ...`

Figura 2.6: Exemplo de laços WHILE e DO-WHILE

Capítulo 3

Compressão Baseada em Fatoração de Instruções

Neste capítulo abordaremos as primeiras tentativas de compressão de byte-code Java que desenvolvemos. As idéias aqui mostradas foram inspiradas nas técnicas de fatoração de árvores de expressão desenvolvidas por Ernst et al [11] e Araujo et al [3]. A contribuição de nossa abordagem foi a de estender esta idéia para compressão de programas em byte-code.

No trabalho de Ernst et al [11], são apresentadas duas formas de compressão. A primeira delas é uma representação executável que possui tamanho próximo ao de programas para plataforma *x86* comprimidos com gzip, e que pode ser interpretada sem a necessidade de descompressão. A *BRISC* (ou *Byte-code RISC*), como é chamada, se utiliza das técnicas de especialização de operandos e combinação de opcodes para conseguir comprimir dados utilizando algoritmos genéricos (ex. Lempel-Ziv [22] e Huffman [19]) e, ainda assim, manter a representação densa e endereçável aleatoriamente, qualidades impossíveis de se obter com tais algoritmos. O trabalho mostra ainda que tal representação aceita geração de código *just-in-time* a uma razão de 2.5MB/sec. A segunda forma de compressão cita um representação comprimida chamada *wire code* cujas razões de compressão alcançam até 21% do tamanho do código binário SPARC equivalente. O *wire code* é o resultado de aplicação de um série de técnicas de compressão e codificação. Os autores inicialmente compilam árvores de código e realizam a fatoração das instruções separando operandos de operadores. Em seguida é aplicada a codificação MTF [6, 10] e o resultado compactado com gzip. Isso é feito para cada seqüência (operandos e operadores) separadamente. O método de compressão descrito nesta seção tem como base uma representação bastante semelhante ao *wire code* aqui apresentado.

Araujo et al [2, 3] mostraram que o número de árvores de instruções distintas em um programa é bem menor que o número total de árvores de expressões no mesmo programa. Neste trabalho foram computadas as freqüências de cada árvore de expressão distinta

usando um conjunto de programas do SPECInt95. As árvores foram ordenadas decrescentemente como função do número de vezes que estas são reutilizadas no programa. Os resultados experimentais revelaram que a freqüência de uma árvore decresce quase que exponencialmente, ou seja uma pequena parcela de árvores de instruções cobre a maior parte das árvores de um programa. Como mostrado na seção 3.3, árvores de expressão formadas a partir de seqüências de byte-code exibem um comportamento semelhante.

A vantagem imediata dos métodos de compressão baseados em fatoração de instruções é o conhecimento que se tem da informação a ser comprimida. Métodos de compressão como Huffman [19] e Lempel-Ziv [22] não se utilizam de nenhuma informação sobre o símbolo a ser comprimido. Nestes casos, o símbolo a ser comprimido é normalmente uma unidade de armazenamento (ex.: bit, byte, word) e não o tipo de dado (ex.: palavras, imagens, instruções).

Linha	Instruções em Byte-Code	Código Associado
1	aload_0	0x2a
2	new 92	0xbb 0x5c
3	dup	0x59
4	aload_0	0x2a
5	invokespecial 99	0xb7 0x63
6	astore 6	0x3a 0x06
7	aload 5	0x19 0x05
8	astore_4	0x4f

Figura 3.1: Trecho de código em byte-code

Para exemplificar o que dissemos acima, considere o trecho de código em byte-code da figura 3.1. No capítulo 2 vimos que algumas instruções têm o tipo do dado codificado na própria instrução. Além disso, algumas instruções também têm codificados seus operandos. Veja a instrução `aload_0` em (1). Esta instrução carrega na pilha uma referência (indicado pela inicial *a*) que se encontra armazenada na variável 0 (zero). Todas essas informações estão contidas no código associado (0x2a). Em outras palavras, nesta instrução 6 bits são utilizados para a instrução em si e 2 bits são utilizados para codificar o operando (de 0 a 3). Observe agora a instrução em `aload 5` em (7). Apesar de ser a mesma instrução, com a mesma semântica e operar no mesmo tipo de dado (referências), esta instrução tem um operando diferente (variável 5) e neste caso, além de ter um código diferente (0x19), passou a ter o operando desvinculado da instrução. Algoritmos como Lempel-Ziv e Huffman comprimem esta seqüência ignorando o significado de cada instrução. O que faremos aqui é, sabendo que as instruções (1) e (7) são a mesma instrução, aproveitar esta redundância de forma a melhorar a compressão do programa.

O que se propõe neste capítulo é um método que procura identificar padrões de repetição no código, tomando-se como símbolos não mais bits, mas expressões formadas a partir de seqüências de instruções do programa (byte-code no caso). Os métodos apresentados aqui tiram proveito das informações como o tipo da instrução, a quantidade e o tipo de seus operandos de modo a escolher os símbolos a serem codificados.

Usando fatoração de instruções nós propomos dois métodos de compressão. No método de casamento de padrões simples, descrito na seção 3.3 nós analisamos como a fatoração de operandos pode ser usada para codificar seqüências (padrões) de operadores e operandos de árvores de expressão. Uma abordagem um pouco mais elaborada, baseada na utilização de prefixos e sufixos dos padrões, é discutida na seção 3.4.

Compressão baseada em fatoração de instruções pode ser dividida em três fases. Primeiramente, as instruções em byte-code de um método são fatoradas. Em seguida a seqüência de instruções é dividida em árvores de expressão, que serão então usadas como símbolos para compressão. A próxima etapa é codificar as expressões de acordo com o número de vezes que cada uma aparece dentro do método. A seguir descrevemos com mais detalhes cada um desses passos.

3.1 Fatoração de Byte-Code

A primeira tarefa é a fatoração das instruções. *Fatorar* uma instrução é a ação de separar o operador dos seus possíveis operandos. Abstraindo-se os operandos, as chances de *matching*, ou seja, as chances de se encontrar seqüências de operadores iguais, aumenta consideravelmente uma vez que as instruções fatoradas formam um conjunto limitado, enquanto que o mesmo não acontece com as instruções não fatoradas. A figura 3.2 mostra um exemplo de fatoração de instruções.

Instruções Não Fatoradas	Operadores	Operandos
aconst_null	iload *	5
fload_3	fload *	3
aaload	aaload	–
getfield 23	getfield *	23

Figura 3.2: Fatoração de instruções

As figuras 3.3 e 3.4 mostram como a fatoração de instruções pode contribuir para o aumento das chances de uma determinada seqüência de instruções se repetir no programa.

Instruções (não fatoradas)	Byte-Code
aconst_null	0x01
astore 5	0x3a **0x05**
new 35 (0x23)	0xbb **0x23**
dup	0x59
invokespecial 10 (0x0a)	0xb7 **0x0a**
astore 6	0x3a **0x06**

Figura 3.3: Trecho A

Instruções (não fatoradas)	Byte-Code
aconst_null	0x01
astore 7	0x3a **0x07**
new 9 (0x09)	0xbb **0x09**
dup	0x59
invokespecial 15 (0x0f)	0xb7 **0x0f**
astore 4	0x3a **0x04**

Figura 3.4: Trecho B

Instruções (fatoradas)	Byte-Code
aconst_null	0x01
astore *	0x3a *
new *	0xbb *
dup	0x59
invokespecial *	0xb7 *
astore *	0x3a *

Figura 3.5: Trechos A e B após fatoração de instruções

Podemos notar que, apesar de ambos os trechos de código A e B utilizarem as mesmas instruções, um método genérico de compressão, como os já citados, não seria capaz de detectar esta semelhança. Isso acontece porque os operandos de cada instrução nos dois trechos são distintos, tornando a seqüência de bytes que os representa parcialmente diferentes. Se efetuarmos a fatoração desses fragmentos antes de aplicar qualquer método de compressão, podemos aumentar a probabilidade de detecção das similaridades entre os dois trechos acima. O resultado da fatoração de instruções em ambos os trechos é uma seqüência única de operadores, mostrado na figura 3.5, que denominamos de *padrão de operadores*. Os bytes em destaque nos trechos A e B representam os operandos que serão fatorados. O asterisco (ou estrela) do trecho fatorado indica que, quaisquer que sejam os operandos, a seqüência de instruções fatoradas será sempre a mesma.

No momento da fatoração é necessário também preservar os operandos de cada operador de forma que consigamos reconstruir cada instrução durante a descompressão. A seqüência de operandos fatorados, aqui chamada de *padrão de operandos*, associada aos trechos A e B são, respectivamente, `[0x05, 0x23, 0x0a, 0x06]` e `[0x07, 0x09, 0x0f, 0x04]`. Desta forma, para os trechos A e B apresentados, teríamos a representação fatorada abaixo. Note que não incluímos as instruções que não possuem operandos (`aconst_null` e `dup`):

Trecho A: [astore, new, invokespecial, astore] [0x05, 0x23, 0x0a, 0x06]
Trecho B: [astore, new, invokespecial, astore] [0x07, 0x09, 0x0f, 0x04]

3.2 Identificação das Árvores de Expressão

Após a fatoração das instruções é necessário definir o tamanho da seqüência de instruções que será usada como símbolo durante a compressão, uma vez que a razão de compressão depende fortemente deste. Note que, devido a combinação de muitas instruções diferentes, seqüências muito longas podem ser raras e portanto exibir freqüências muitos baixas. Por outro lado, seqüências curtas podem simplesmente aumentar em muito o número de símbolos distintos e portanto resultar em palavras de código longas inviabilizando a compressão. Ao invés de experimentar com seqüências de instruções de tamanhos variados, como é feito em outros estudos [15, 21], nós decidimos explorar uma característica própria da JVM para escolher a seqüência símbolo. Por ser a JVM uma máquina a pilha, toda sentença ou expressão em Java é compilada para uma seqüência de instruções em byte-code na qual o estado inicial da pilha é igual ao seu estado final após esta seqüência ter sido executada. Em outras palavras, dado um estado inicial da pilha de operandos da JVM (normalmente vazia no início de um programa ou método), ao fim da execução das instruções em byte-code correspondentes à uma sentença ou expressão Java, a pilha

de operandos se encontrará na mesma situação que antes da execução. Para exemplificar, considere o exemplo da figura 3.6 abaixo:

Linha	Instrução	Situação da Pilha de Operandos (após execução)
1	[situação inicial]	()
2	new 5	(obj5)
3	dup	(obj5, obj5)
4	aload_0	(ref0, obj5, obj5)
5	iconst_0	(const0, ref0, obj5, obj5)
6	aaload	(ref0[0], obj5, obj5)
7	invokespecial 17	(obj5)
8	astore_3	()

Figura 3.6: Exemplo de variação da pilha de operandos da JVM

Na figura 3.6 nós mostramos um trecho de código e assumimos que a pilha de operandos da JVM está vazia (representada com parênteses vazios na linha 1) imediatamente antes da execução. À medida que as instruções são executadas, a pilha tem seus valores alterados. A instrução na linha 2 aloca um novo objeto, cujo índice na tabela de constantes é 5, e coloca uma referência ao mesmo no topo da pilha de operandos. Neste momento, a pilha contém apenas esta referência e tem seu conteúdo representado por (obj5) ou objeto cujo índice da classe é 5. A instrução seguinte (dup, linha 3) duplica o conteúdo do topo da pilha e por conseguinte deixa a mesma no estado (obj5, obj5). As instruções são executadas até que, na linha 8, astore_3 remove da pilha a referência ao objeto, levando a pilha de operandos a ficar novamente vazia. Este comportamento pode ser observado em qualquer outro tipo de expressão ou sentença Java. Devido a essa particularidade, escolhemos o conjunto de instruções que compõem uma sentença ou expressão Java como nosso símbolo para compressão, que passará a ser chamado daqui por diante de *expressão*.

Os compiladores, enquanto programas, se comportam de forma similar todas as vezes que são alimentados com entradas parecidas. Em outras palavras, possuem comportamento determinístico. Assim, dadas seqüências parecidas em linguagem de alto nível, espera-se que o código gerado pelo compilador seja também bastante parecido. Acrescenta-se a isso o fato de que muitas linguagens de alto nível, como C/C++ e Java, possuem um conjunto limitado de estruturas de controle que se repetem bastante no decorrer de um programa. Estruturas como *if-then-else*, *while*, *do-while* e *for* aparecem repetidas vezes em um programa típico. No caso específico da JVM ainda existe um outro fator. Além do compilador Java gerar códigos similares, a JVM também trabalha de forma determinística. Logo, na presença de trechos de código semelhantes, a JVM fará uso semelhante da pilha

de operandos. Estas conclusões reforçam a idéia de se usar expressões formadas a partir da pilha de operandos como símbolo de compressão.

3.3 Detecção de Expressões Repetidas

A terceira fase deste método de compressão é chamada de detecção de expressões e é responsável pela codificação do byte-code em uma representação comprimida. A qualidade do algoritmo desta última fase determina a qualidade do algoritmo de compressão como um todo. Nesta fase procuramos pelas repetições das expressões construídas na etapa anterior e codificamos tais expressões com base no tamanho e no número de vezes que cada uma aparece repetida dentro do código. Neste algoritmo, nós utilizamos um método bem simples de casamento de padrões (aqui chamado de *SimpleExpr* ou Simple Expression), que consiste numa simples comparação entre as expressões encontradas. As expressões são então codificadas de acordo com a contribuição que têm para o tamanho do código. Esta contribuição, aqui chamada de *peso*, é calculada multiplicando-se seu tamanho pelo número de vezes que o padrão se repete dentro do código. Depois as expressões são codificadas de acordo com seu peso: expressões com pesos maiores recebem códigos menores.

Para exemplificar esta última fase, considere o método descrito na figura 3.7 abaixo:

```
aload_0
getfield 159
invokevirtual 60
aload_1
getfield 146
invokevirtual 61
aload_0
getfield 45
invokevirtual 110
aload_1
getfield 67
invokevirtual 111
iadd
ireturn
```

Figura 3.7: Método em byte-code Java

O trecho de código acima é a representação em byte-code de um método em Java. É importante saber alguns detalhes a respeito deste trecho. O campo 159 do objeto referenciado pela variável 0 e o campo 146 do objeto referenciado pela variável 1 são inteiros. O

método invocado número 60 (variável 0) e 61 (variável 1) aceitam um parâmetro único inteiro e não retornam valor. Os métodos 110 (variável 0) e 111 (variável 1) também aceitam um único parâmetro inteiro, mas diferem dos anteriores por retornarem um valor inteiro. Assim, seguindo os passos descritos anteriormente, primeiro devemos fatorar todas as instruções. Em seguida, devemos separar as instruções em blocos chamados expressões. A figura 3.8 mostra o mesmo trecho de código já fatorado e dividido em expressões.

```
e1: (3)              e2: (3)              e3: (8)
aload *              aload *              aload *
getfield *           getfield *           getfield *
invokevirtual *      invokevirtual *      invokevirtual *
                                          aload *
                                          getfield *
                                          invokevirtual *
                                          iadd
                                          ireturn
```

Figura 3.8: Método em byte-code fatorado e separado em expressões — SimpleExpr

Os valores $e1$, $e2$ e $e3$ representam as três expressões encontradas no trecho anterior. Os valores entre parênteses representam o tamanho de cada expressão (em número de instruções — a maioria das instruções em byte-code Java ocupa 1 byte). Note que as expressões $e1$ e $e2$ são exatamente iguais quando comparadas depois da fatoração de instruções. O algoritmo de comparação de expressões — SimpleExpr — detecta esta semelhança. O próximo passo é calcular o peso que cada expressão tem dentro do código. O peso, como dito anteriormente, é calculado multiplicando-se o tamanho da expressão pela sua freqüência. Assim, a expressão $e1$ (ou $e2$) tem peso 6 (duas aparições com tamanho 3) e a expressão $e3$ tem peso 8. A idéia é agora codificar as expressões encontradas com base em seu peso. Assim como no algoritmo de Huffman [19], nós utilizamos códigos menores para as expressões com pesos maiores.

Utilizamos uma *hash table* para armazenar as expressões encontradas. Cada elemento desta *hash table* é um estrutura composta pela própria expressão, um contador que registra o número de repetições bem como uma variável que representa seu peso. No final do processo de extração de expressões, os elementos da *hash table* são transferidos para uma lista ordenada decrescentemente em relação ao peso. As expressões são então codificadas utilizando-se Huffman e gravadas em arquivo.

É importante salientar que todo este processo de compressão realizado sobre os operadores também é aplicado aos operandos. Como já mencionado, as operações aqui apre-

sentadas são aplicadas separadamente às seqüências de operandos e operadores.

As figuras 3.9 e 3.10 mostram a contribuição dos padrões de operadores e operandos para o tamanho dos programas.

Nas figuras nota-se que a distribuição tende a ser mais exponencial à medida que o tamanho do programa cresce. Infelizmente, devido à natureza da linguagem Java, os projetos feitos nesta linguagem tendem a ser compostos de várias classes, em sua maioria de tamanho reduzido. Assim, a compressão isolada das classes utilizando o método SimpleExpr se mostrou ineficiente, produzindo razões de compressão baixas. As razões para isso são aquelas explicitadas anteriormente. As várias classes Java, quando compiladas, resultam em vários arquivos de classe Java (código compilado) de tamanho reduzido, não permitindo que os algoritmos de compressão encontrem redundâncias presentes em arquivos distintos. Mostramos, portanto, que tal abordagem não apresentou resultados tão bons quanto àqueles obtidos em código binário nativo para plataforma SPARC, apresentados em [3].

Na seção 3.5 nós mostramos os resultados obtidos a partir deste método de compresão. A tabela 3.2 (página 36) ilustra como o SimpleExpr se compara ao LZW e ao método PrefixExpr que será descrito na próxima seção.

3.4 Casamento de Prefixos e Sufixos

No algoritmo SimpleExpr descrito anteriormente, as expressões encontradas eram simplesmente comparadas umas com as outras a fim de se encontrar o número de repetições de cada uma delas. Um *matching* somente ocorria se todas as instruções de duas expressões fossem exatamente iguais. Por outro lado, observamos que, em muitos casos, expressões inteiras apareciam dentro de outras expressões maiores. Uma alternativa a nossa abordagem inicial era portanto detectar expressões que eram sub-expressões de outras expressões. Com isto estaríamos aumentando o número de expressões repetidas mas também aumentando o tempo de execução de nosso algoritmo, visto que a detecção exata de padrões é uma problema NP-difícil [16]. Uma alternativa à solução exata é a de se limitar o tamanho das sub-expressões e/ou o seu alinhamento dentro de uma expressão maior. Assim sendo, introduzimos uma variação em SimpleExpr visando capturar subexpressões comuns. Neste novo algoritmo — denominado *PrefixExpr* — a fase de detecção de expressões comuns é modificada de modo a detectar se os prefixos e sufixos das expressões podem conter outras expressões menores. Assim sendo, uma expressão como $e3$, apresentada na figura 3.8, passa agora a ter o seu conteúdo analisado e seus prefixos e sufixos contabilizados no processo de identificação de expressões comuns. Neste algoritmo, nós consideramos todas as expressões de tamanho superior a três e construímos todos os prefixos e sufixos de tamanho mínimo três e tamanho máximo igual ao tamanho da

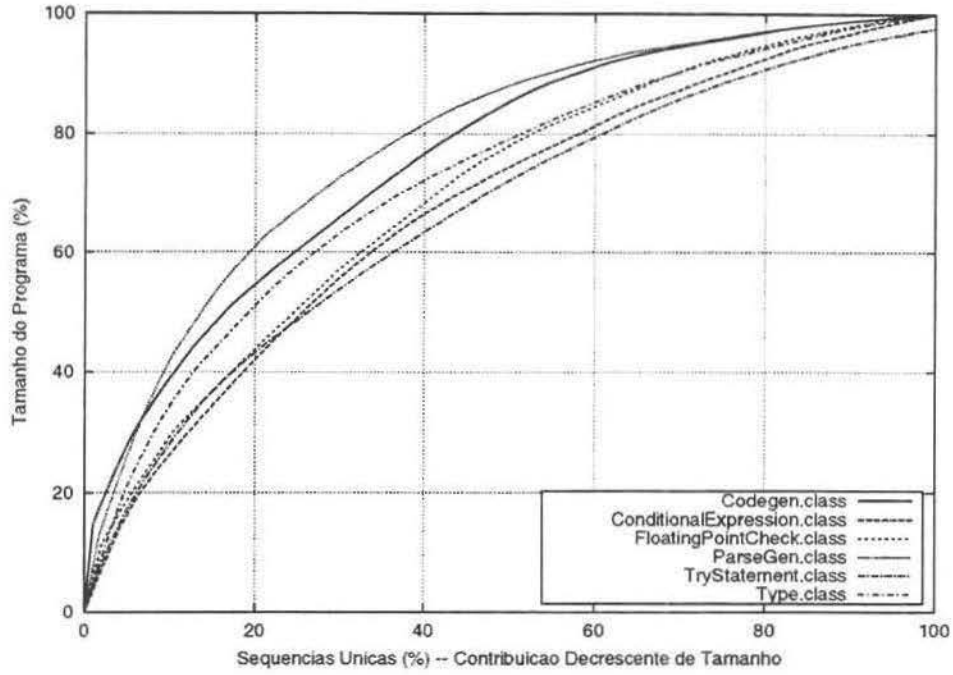


Figura 3.9: Contribuição dos padrões de operadores para o tamanho do código — Método SimpleExpr

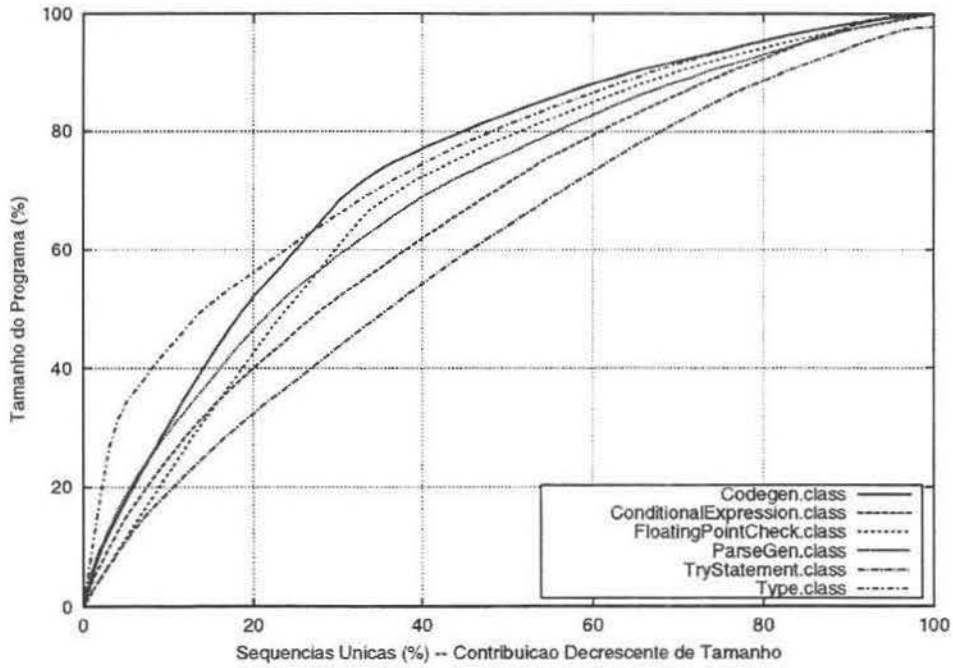


Figura 3.10: Contribuição dos padrões de operandos para o tamanho do código — Método SimpleExpr

expressão menos três. A figura 3.11 mostra o resultado da aplicação deste novo algoritmo no mesmo trecho de código da figura 3.7 (página 26).

O algoritmo de detecção de expressões do método SimpleExpr é bastante trivial e foi omitido propositalmente. Já o método PrefixExpr é um pouco mais elaborado. Como mencionado anteriormente, além de considerar também as próprias expressões, o algoritmo contabiliza os prefixos e sufixos da mesma. O pseudo-código deste é mostrado no algoritmo 1. Este algoritmo recebe dois argumentos. O primeiro deles é uma lista encadeada dos opcodes de uma expressão (opl) e o segundo, o tamanho em número de instruções de tal lista (size). Dados uma lista, a posição inicial e a posição final dentro da lista, o método auxiliar `buildPattern()`, constrói um padrão (ou uma subexpressão) de acordo com os argumentos recebidos. Nas linhas 3 e 4 registramos as expressões de operadores e operandos, respectivamente, que foram construídas na linha 2. A partir da linha 6, caso a expressão em questão tenha tamanho maior que o tamanho mínimo necessário para se realizar o processamento de seus prefixos e sufixos (tamanho mínimo estipulado em 3 instruções — `minimumSize`), nós construímos todos os prefixo (linhas 8–12) e sufixos (linhas 15–19) com tamanho mínimo 3 e tamanho máximo igual ao tamanho da expressão em questão menos 3.

```
e1: (3)
aload *
getfield *
invokevirtual *

e2: (3)
aload *
getfield *
invokevirtual *

e3: (8)
aload *
getfield *
invokevirtual *
aload *
getfield *
invokevirtual *
iadd
ireturn

e3.1: (3)
aload *
getfield *
invokevirtual *

e3.2: (4)
aload *
getfield *
invokevirtual *
aload *

e3.3: (5)
aload *
getfield *
invokevirtual *
aload *
getfield *

e3.4: (6)
aload *
getfield *
invokevirtual *
aload *
getfield *
invokevirtual *

e3.5: (7)
aload *
getfield *
invokevirtual *
aload *
getfield *
invokevirtual *
iadd

e3.6: (3)
invokevirtual *
iadd
ireturn

e3.7: (4)
getfield *
invokevirtual *
iadd
ireturn

e3.8: (5)
aload *
getfield *
invokevirtual *
iadd
ireturn

e3.9: (6)
invokevirtual *
aload *
getfield *
invokevirtual *
iadd
ireturn

e3.10: (7)
getfield *
invokevirtual *
aload *
getfield *
invokevirtual *
iadd
ireturn
```

Figura 3.11: Trecho de código em byte-code fatorado e separado em expressões — Método PrefixExpr

Algoritmo 1 Pseudo-código de identificação de expressões — PrefixExpr

```
1   void exprDetect(OpcodeList opl, int size) {
2       Pattern expr = buildPattern(opl, 1, size);
3       registerOperatorPattern();
4       registerOperandPattern();
5       // Monta prefixos e sufixos caso o tamanho da expressão seja > 3
6       if (size > minimumSize) {
7           // Contabiliza prefixos
8           for(int i = minimumSize; i < size - minimumSize + 1; i++) {
9               Pattern p = buildPattern(opl, 1, i);
10              registerOperatorPattern(p);
11              registerOperandPattern(p);
12          }
13
14          // Contabiliza sufixos
15          for(int i = size - minimumSize + 1; i > minimumSize; i--) {
16              Pattern p = buildPattern(opl, i, size - i + 1);
17              registerOperatorPattern(p);
18              registerOperandPattern(p);
19          }
20      }
21  }
```

A primeira complicação encontrada neste algoritmo é o trabalho computacional adicional de se identificar os prefixos e sufixos das expressões. Além disso, existe também um outro complicador que surge no momento de decidir que expressões farão parte do dicionário na codificação de Huffman. No momento que escolhemos uma sub-expressão (prefixo ou sufixo) como elemento do dicionário, temos também que adicionar no dicionário seu complementar em relação à expressão. Isto acontece porque corremos o risco de deixar de considerar alguns trechos do código. Na figura 3.11 são mostradas as expressões identificadas e seus respectivos tamanhos. A tabela 3.1 foi construída a partir destas informações.

Note que expressões $e1$, $e2$ e $e3.1$ são idênticas e por isso foram contabilizadas como uma só. Como podemos observar, a expressão representada por estas tem peso 9 e certamente esta expressão constará no dicionário. Contudo, uma vez que o algoritmo considerou a inclusão da expressão $e3.1$, temos também que incluir no dicionário sua expressão complementar em relação à expressão $e3$ de modo que a expressão $e3.8$ também

Expressão	Peso
e1, e2, e3.1	9
e3	8
e3.2	4
e3.3	5
e3.4	6
e3.5	7
e3.6	6
e3.7	4
e3.8	5
e3.9	6
e3.10	7

Tabela 3.1: Expressões e seus respectivos tamanhos

conste do dicionário. Deduzimos então que a escolha das sub-expressões que constarão no dicionário é função do cálculo da maior freqüência da sub-expressão em si e do seu complementar. Da mesma forma que no algoritmo SimpleExpr, as expressões são ordenadas decrescentemente segundo seu peso e são posteriormente codificadas utilizando-se Huffman.

Ainda há espaço para melhorias no algoritmo de detecção de expressões comuns. No entanto, à medida que tornamos o algoritmo mais especializado, aumentamos sua complexidade. É fácil perceber que o algoritmo ideal de detecção de expressões, seguindo essa mesma idéia, deveria considerar toda e qualquer combinação de instruções do programa, desde combinações com uma única instrução até uma combinação com todas as instruções (o próprio programa). No entanto, as complexidades de tempo e espaço para tal algoritmo seriam bastante elevadas. Apesar de ser de fácil implementação, tal algoritmo certamente não seria competitivo o suficiente para justificar a sua utilização.

Assim como no método SimpleExpr, apresentamos nas figuras 3.12 e 3.13 a contribuição dos padrões de operandos e operadores para o tamanho do código quando o método PrefixExpr é usado.

3.5 Resultados Experimentais

Nesta seção nós mostramos alguns dos número obtidos utilizando-se os dois métodos descritos neste capítulo. Para tanto utilizamos alguns arquivos que compõe o Benchmark JVM98 além de uma classe proprietária chamada *Codegen.class*, escolhida para fazer os testes antes da aquisição do Benchmark JVM98. Utilizamos também o programa gzip versão 1.2.4 (18 Aug 93) com a flag *best compression* ligada. A plataforma de desenvolvi-

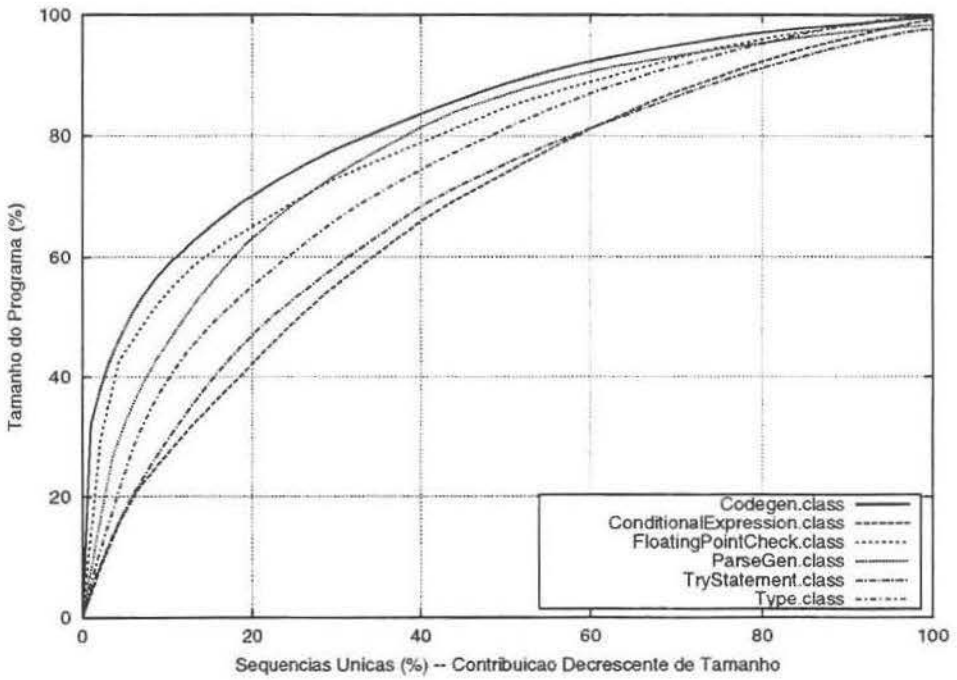


Figura 3.12: Contribuição dos padrões de operadores para o tamanho do código — Método PrefixExpr

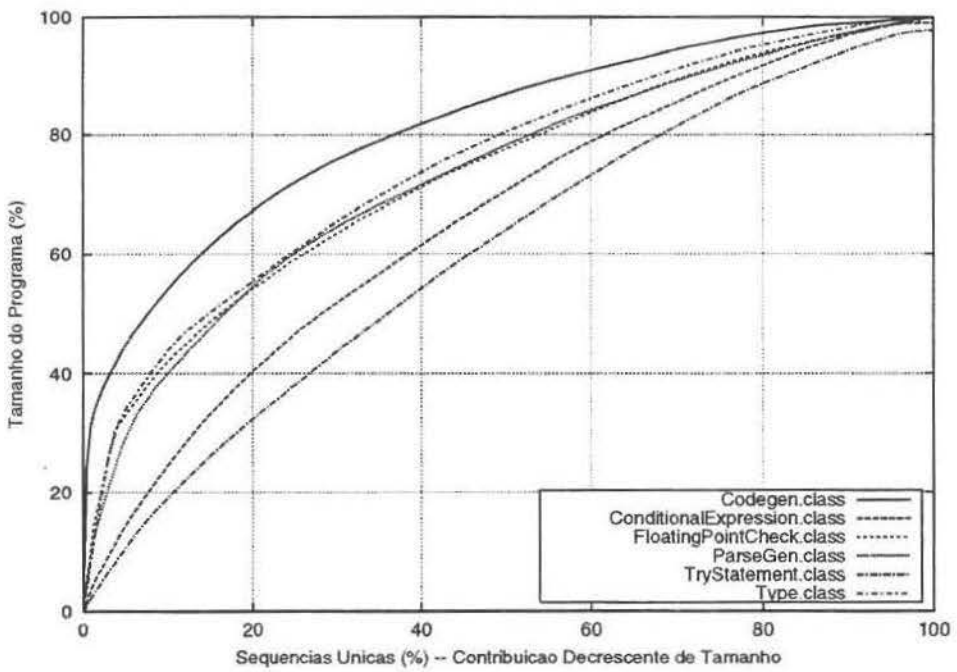


Figura 3.13: Contribuição dos padrões de operandos para o tamanho do código — Método PrefixExpr

mento e de testes foi o Slackware GNU/Linux versões 4.0 e 7.0 rodando num processador Intel Pentium II de 416MHz e 128MB de memória RAM.

A tabela 3.2 mostra os resultados obtidos aplicando-se os algoritmos SimpleExpr e PrefixExpr em algumas classes que compõe o Benchmark JVM98 além da classe *Codegen.class*. Também são mostrados os resultados da aplicação do compressor gzip nas mesmas classes para efeito comparativo. Os valores entre parênteses representam o tamanho do programa comprimido em relaçao ao tamanho do programa original (100%).

Todos os valores da tabela estão em bytes. Podemos notar uma pequena melhoria na razão de compressão quando comparamos o método SimpleExpr com o método PrefixExpr. Isto pode ser explicado pelo uso do algoritmo mais elaborado de detecção de expressões. Contudo, percebemos uma perda de desempenho quando trabalhamos com classes muito pequenas. Apesar dos melhores resultados, o método PrefixExpr ainda não se mostra competitivo a algoritmos genéricos como o LZW utilizado pelo programa gzip.

Na tabela 3.3 nós encontramos os tempos de execução do compressor para os métodos SimpleExpr e PrefixExpr em todos as classes presentes na tabela 3.2. Também incluímos o tempo de inicialização da JVM na plataforma de teste. Todos os resultados são uma média de cinco execuções. Na última coluna da tabela encontramos o desvio padrão calculado para o tempo médio apresentado. Podemos notar que, à medida que o tamanho do programa cresce linearmente, a discrepância entre o tempo de execução para os métodos SimpleExpr e PrefixExpr aumenta exponencialmente. Este comportamento, mostrado na figura 3.14, era esperado uma vez que o método PrefixExpr adiciona complexidade à etapa de detecção de expressões comuns dos métodos baseados na fatoração de árvores de expressão vistos neste capítulo. Não incluímos neste gráfico o tempo do algorimo gzip pois o mesmo foi implementado em C e roda em código nativo, enquanto que os métodos SimpleExpr e PrefixExpr estão implementados em Java, tornando inválida qualquer comparação direta entre eles.

Programa	Tamanho	LZW	*SimpleExpr*	*PrefixExpr*
TryStatement.class	1052	699 (66,44%)	1031 (97,99%)	1007 (95,63%)
ConditionalExpression.class	1077	543 (50,41%)	968 (89,83%)	941 (87,33%)
FloatingPointCheck.class	1131	480 (42,44%)	1089 (96,28%)	914 (71,97%)
Type.class	1238	740 (59,77%)	1226 (98,97%)	1190 (96,05%)
ParseGen.class	3431	1193 (34,77%)	2386 (69,51%)	2018 (58,80%)
Codegen.class	4822	1465 (30,38%)	4024 (83,44%)	2837 (58,83%)

Tabela 3.2: Comparação dos resultados obtidos com casamento de padrões

Classe	Tamanho	Método	Tempo (s)	Desvio
Inicialização da JVM	6	—	0,192	$5,5 \times 10^{-4}$
TryStatement.class	1052	SimpleExpr	1,090	$1,9 \times 10^{-3}$
		PrefixExpr	1,203	$1,3 \times 10^{-3}$
ConditionalExpression.class	1077	SimpleExpr	1,118	$1,3 \times 10^{-3}$
		PrefixExpr	1,371	$1,2 \times 10^{-3}$
FloatingPointCheck.class	1131	SimpleExpr	0,999	$1,9 \times 10^{-3}$
		PrefixExpr	1,393	$1,1 \times 10^{-3}$
Type.class	1238	SimpleExpr	1,128	$1,6 \times 10^{-3}$
		PrefixExpr	1,338	$1,1 \times 10^{-3}$
ParseGen.class	3431	SimpleExpr	2,818	$2,8 \times 10^{-3}$
		PrefixExpr	4,368	$2,3 \times 10^{-3}$
Codegen.class	4288	SimpleExpr	4,280	$1,1 \times 10^{-2}$
		PrefixExpr	10,072	$1,7 \times 10^{-2}$

Tabela 3.3: Tempos de execução (em segundos) para os métodos SimpleExpr e PrefixExpr

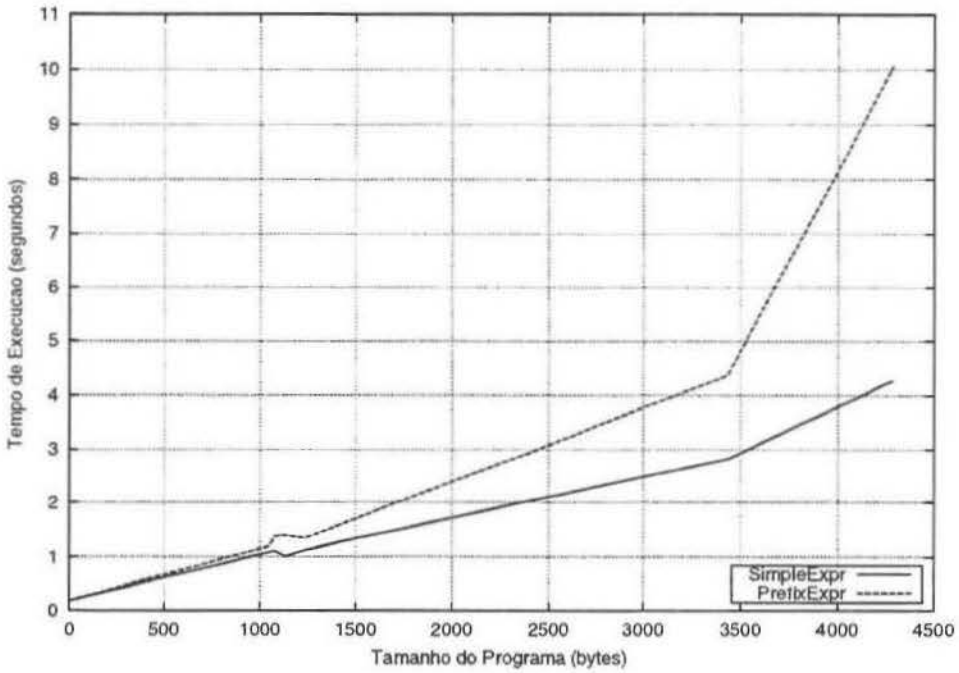


Figura 3.14: Gráfico Tamanho do program x Tempo de execução

Capítulo 4

Compressão Baseada em Árvores de Sintaxe

Neste capítulo nós descrevemos um método alternativo de compressão, que resulta em boa compressão. Este método usa informações sobre a estrutura da linguagem e do compilador Java para realizar uma decompilação parcial do código objeto em uma representação intermediária que é então comprimida. Ao mesmo tempo em que resulta em boas razões de compressão, este método dispensa o algoritmo de casamento de padrões usado em Fatoração de Árvores, tornado-o muito mais rápido que este.

O método de compressão baseado em árvores de sintaxe abstrata pode ser descrito em duas grandes etapas: *decompilação* (seção 4.2) e *codificação* (seção 4.3). A decompilação compreende a transformação do código objeto Java (arquivo de classe) em uma representação intermediária e na criação de estruturas auxiliares. A codificação vem em seguida e tem por objetivo codificar as árvores de sintaxe abstrata e aplicar o algoritmo de compressão nas mesmas.

4.1 Java Abstract Syntax (JAS)

A nossa sintaxe abstrata, denominada *Java Abstract Syntax* (JAS) é baseada na Pilha de Operandos Java e é altamente dependente da Tabela de Constantes. As decisões sobre o formato das instruções em JAS foram tomadas com o objetivo de tornar programas decompilados em JAS o mais compacto possíveis. JAS tem o poder de representação do programa fonte em Java e portanto resulta em uma representação menos complexa (menor) que o código de máquina em byte-code correspondente. Em outras palavras, dado um fragmento de byte-code, podemos sempre garantir que as ASA recuperadas terão tamanho igual ou, mais freqüentemente, menor que o fragmento original. Isso acontece por causa das simplificações e generalizações resultantes do processo de decompilação, descrito na

seção 4.2. Basicamente, JAS é composta do que chamamos *expressões sintáticas Java* (ou simplesmente *expressões*), que serão os símbolos básicos utilizados durante a compressão. Expressões são formadas por um conjunto de instruções de uma seqüência de byte-code. O algoritmo de detecção de expressões, já discutido na seção 3.2, página 24, é também utilizado aqui para separar o programa em símbolos a comprimir. A tabela 4.1 mostra a relação entre algumas das instruções em byte-code e suas respectivas representações em JAS. É importante ressaltar que nesta tabela estão presentes algumas das instruções que possuem equivalência direta do Java para JAS bem como algumas instruções JAS que aglomeram várias instruções em byte-code (ex.: ALLOC e SHORTC).

4.1.1 As Instruções da JAS

A escolha do mapeamento entre byte-code e instruções em JAS teve como objetivo primordial capturar blocos de byte-code que se repetem freqüentemente dentro de um programa, bem como generalizar certas instruções. Uma maneira de se atingir isto é abstraindo os tipos das instruções. Desta forma, podemos construir versões genéricas dessas instruções e, ao usá-las, estaremos aumentando a redundância do código. Aumentando a redundância estamos também aumentando as razões de compressão. A seguir descrevemos as instruções JAS da tabela 4.1, procurando justificar como elas contribuem para uma representação mais compacta de um programa.

Instruções ALLOC

O objetivo da escolha da instrução ALLOC foi simplificar o bloco de instruções para alocação de objetos, visto que a seqüência de instruções em byte-code usada para executar esta tarefa repete-se freqüentemente dentro de um programa Java. Esta seqüência é formada pelo opcode *new* seguido do opcode *dup*, os argumentos do construtor do objeto e a invocação do construtor (opcode *invokespecial*), como mostrado na figura 4.1. Com exceção dos argumentos do construtor, todas as instruções em byte code podem ser substituídas pela instrução ALLOC em JAS.

O trecho de código byte-code da figura 4.1 aloca uma instância do objeto cujo índice é 88, inicializando-o em seguida. A inicialização do objeto corresponde à invocação de seu construtor passando o byte 16 como parâmetro e deixando no topo da pilha uma referência para o mesmo. Este trecho pode ser traduzido para uma única instrução em JAS:

```
ALLOC(88, CONST(16))
```

Esta instrução engloba em si a chamada à instrução dup, bem como a invocação do construtor. Isto é feito mediante uma pesquisa na Tabela de Constantes que nos permite

Byte-Code	JAS	Byte-Code	JAS
nop	NOP	tableswitch lookupswitch	SWITCH()
aconst_null	CONST(NULL)		
iconst_*c*	ICONST(c)	getstatic getfield	GET ou IGET
lconst_*c*	LCONST(c)		
fconst_*c*	FCONST(c)	putstatic putfield	PUT ou IPUT
dconst_*c*	DCONST(c)		
bipush *c* sipush *c*	CONST(c)	invokevirtual invokespecial invokestatic invokeinterface	INVOKE(type)
ldc *c* ldc_w *c* ldc2_w *c*	CONST(c)		
*x*aload	ALOAD	new multinewarray	ALLOC()
*x*store_*c* *x*store *c*	STORE(c)	arraylength	ARRAYLENGTH()
		athrow	THROW()
*x*store	ASTORE	checkcast	CHECKCAST()
*x*add *x*sub *x*mul *x*div *x*rem *x*neg *x*shl *x*shr *x*ushr *x*and *x*or *x*xor iinc	OPER(type)	instanceof	INSTANCEOF()
		monitorenter monitorexit	MONITOR(type)
		wide *instr*	WIDE(instr)
		ret *x*return return	RETURN()
		*x*cmp *x*cmpl *x*cmpg	OPER()
		if*cond* if_*x*cmp*cond* ifnull / ifnonnull	CJUMP()
*x*2*y*	sem equivalente	sem equivalente	SHORTC

Tabela 4.1: Relação entre instruções em byte-code e JAS

```
[...]
new            88  ; aloca novo objeto na pilha
dup                ; duplica referência (parâmetro para construtor)
bipush         16  ; carrega argumento para construtor
invokespecial  76  ; chamada ao construtor
[...]
```

Figura 4.1: Alocação de objetos em byte-code

identificar o construtor de uma classe tendo apenas o índice para a descrição da mesma. Desta forma podemos economizar alguns bytes em cada instanciação de classe, ação bastante comum em programas Java.

Instruções IGET e IPUT

Diferentemente da linguagem Java ou C++, que faculta o uso da palavra reservada *this* quando não há ambigüidade, o acesso a um atributo a de um objeto X ($X.a$) em byte-code sempre requer um ponteiro como referência ao objeto em questão (ver figura 4.2).

```
aload      X ; carrega referência ao objeto X
getfield   a ; carrega X.a na pilha de operandos
```

Figura 4.2: Referência ao atributo a do objeto X em byte-code

Como a maioria dos acessos a atributos em linguagens orientadas a objetos são feitos relativamente ao próprio objeto (acessos a um atributo a do objeto X a partir de métodos da classe que define o objeto X), a instrução *aload* que antecede o acesso ao atributo é desnecessária. Desta forma, nós decidimos criar uma versão especial em JAS das instruções byte-code de acesso a atributos *putfield* e *getfield*. A instrução *getfield* retira da pilha uma referência e coloca de volta na pilha o atributo representado pelo operando. *putfield* funciona de forma semelhante, mas salva no atributo o conteúdo do topo da pilha. Na maior parte das vezes, a referência utilizada por estas instruções é carregada na pilha através da instrução *aload_0*, o equivalente ao ponteiro *this* em C++ ou Java. A figura 4.3 mostra o uso da instrução IPUT em JAS, que sintetiza o acesso para escrita do campo a do objeto X. De maneira similar a instrução IGET pode ser usada para simplificar o acesso de leitura aos campos do objeto corrente.

Java	Byte-Code	JAS

`X.a = 0;`	`iconst_0`
	`aload_0`
	`putfield 1`
		`IPUT(1, CONST(0))`

Figura 4.3: Acesso a atributos do objeto corrente

Instruções INVOKE

As instruções da família INVOKE são utilizadas para as chamadas de procedimento e função. No entanto, como já mostrado no capítulo 2, há quatro variações desta instrução em byte-code, uma para cada tipo de invocação de método, seja ela estática, virtual, de interface ou especial. Achamos que seria interessante se pudéssemos abstrair o tipo da invocação e passássemos a utilizar apenas uma instrução genérica denominada simplesmente de INVOKE. Contudo, somente com a informação da Tabela de Constantes não é possível fazer a distinção do tipo de instrução INVOKE a utilizar. O uso de variáveis e métodos externos à classe em questão nos impede de fazer tal distinção uma vez que suas informações não estão disponíveis. Por exemplo, se em um método da classe *A* invocamos um método *b()* pertencente à classe *B*, teremos que utilizar métodos especializados para a invocação de *b()* uma vez que suas informações estão disponíveis apenas no arquivo de classe que define a classe *B*. Para casos com este, fizemos uso de instruções de invocação de métodos especializados em JAS que correspondem às mesmas instruções em byte-code (invokespecial, invokevirtual, etc). Para invocações de métodos locais à classe (ex.: invocação de um método *a()* pertencente à classe *A*), nós utilizamos a instrução INVOKE genérica.

A figura 4.4 mostra exemplos de dois trechos de código (duas invocações de método) em byte-code e seu correspondente em JAS. Note como tivemos que utilizar instruções INVOKE diferentes para cada um deles.

Note que estamos instanciando o objeto `objA` e efetuando uma invocação de método. No primeiro exemplo (figura 4.4b) invocamos o método `a()`, que é um método da própria classe `A` cujas informações estão disponíveis em seu arquivo de classe. Por isso fomos capazes de utilizar a versão genérica da instrução INVOKE. Já no segundo exemplo (figura 4.4c), invocamos um método `b()`, pertencente ao objeto `objB` (classe `B`) que por sua vez pertence ao objeto `objA`. Como o método `b()` não está descrito no arquivo de classes de `A`, não temos como avaliar o tipo de invocação para tal método. Por isso fizemos uso da versão JAS especializada da instrução INVOKE.

Classe A	Classe B

public class A {
  B objB;
  void a();
}
	public class B {
	  static void b();
	}

(a)

Linha	Java	Byte-Code	JAS
1	void main() {	new 3	STORE(1,
2	A objA;	dup	ALLOC(3)
3		invokespecial 11	)
4	objA.a();	astore_1	INVOKE(
5	}	aload_1	LOAD(1),
6		invokevirtual 12	12
7			)

(b)

Linha	Java	Byte-Code	JAS
1	void main() {	new 3	STORE(1,
2	A objA;	dup	ALLOC(3)
3		invokespecial 11	)
4	objA.objB.b();	astore_1	INVOKE(ST,
5	}	aload_1	LOAD(1),
6		invokestatic 13	12
7			)

(c)

Figura 4.4: (a) Descrição de duas classes A e B. (b) Exemplo de invocação de método interno à classe em JAS. (c) Exemplo de invocação de método externo à classe em JAS

Instruções LOAD e STORE

Devido a uma característica intrínseca da JVM, que mantém informações a respeito do tipo de cada constante ou variável de cada método de uma classe, nós fomos capazes de criar instruções genéricas para o carregamento ou armazenamento de valores da pilha de operandos. Em byte-code Java, normalmente referenciamos o tipo de um operando quando o carregamos na pilha ou o salvamos em uma variável (figura 4.5). Esta mesma característica da máquina virtual determina que todas as variáveis em byte-code devem ter seu valor inicializado antes de qualquer uso. Isso torna nossa tarefa ainda mais fácil pois, uma vez inicializada uma variável, saberemos seu tipo e podemos manter este registro até o final do método em questão.

```
aload    4 ; carrega referência a um objeto de 4 para pilha
dstore_2   ; salva o topo da pilha (tipo double), em 2
lload_0    ; carrega um valor long de 0 para a pilha
```

Figura 4.5: Tipos são codificados diretamente nas instruções do byte-code

Em nossa representação intermediária, nós criamos versões generalizadas de instruções como load e store. Desta forma, todas as instruções da forma *xload* seriam substituídas por uma instrução JAS simples *LOAD*, e todas *xstore*, por *STORE*. No entanto, não podemos perder a informação a respeito do tipo de cada variável. Precisaremos desta informação no momento da descompressão. Sabendo disso, nós procuramos manter sempre o tipo de cada variável implicitamente definido em alguma instrução. Para atribuições, por exemplo, a constante ou variável do lado direito da expressão de atribuição carrega sempre seu tipo associado. Além disso, sabemos o tipo de uma variável no momento de sua inicialização. Para chamadas de métodos existe sempre um descrição do método na Tabela de Constante do arquivo de classe Java. Desta descrição podemos extrair o número de parâmetros e seus tipos, bem como o tipo do retorno do método. O exemplo da figura 4.6 mostra como isso é feito. No trecho de atribuição (à esquerda), vemos a variável 0 sendo atribuída o valor NULL. Daí determinamos o tipo da variável 0 como sendo uma referência. Quando a atribuição `STORE(1, 0)` é executada, assumimos que o tipo da variável 1 também será uma referência. Do exemplo de chamada de método, podemos determinar os tipos das variáveis 1, 3 e 4 através de uma consulta à Tabela de Constantes. O descritor para o método 13 informa que o método 13 não possui valor de retorno (void) e mostra que este recebe 3 argumentos: um inteiro, outro inteiro e um double, respectivamente.

Atribuição	Chamada de Método
`STORE(0, CONST(NULL))`	`Constant Pool: V 13(IID)`
`STORE(1, 0)`	`INVOKE 13(1, 3, 4)`

Figura 4.6: Associação implícita de tipos

Instrução OPER

Todas as instruções aritméticas e lógicas em byte-code foram mapeadas numa única instrução JAS, chamada OPER. As instruções aritméticas e lógicas em byte-code (adição, multiplicação, negação, etc) foram agrupadas pois todas resultam num valor que será usa-

do como operando em outras instruções. Mapeando todas essas instruções numa única instrução JAS estamos simplificando a ASA assim como aumentando a redundância do código, uma vez que a instrução OPER codifica o tipo dos operandos de forma implícita, assim como nas instruções LOAD e STORE.

Byte-Code Java	JAS
	STORE(2,
iload_2	OPER(SUB,
iload_3	LOAD(2),
isub	LOAD(3)
istore_2	)
	)

Figura 4.7: Exemplo de uso da instrução OPER

4.2 Decompilação de Byte-code em JAS

Podemos subdividir a etapa de decompilação em algumas fases. A primeira delas divide o código objeto em blocos básicos, criando uma lista encadeada cujos elementos são os próprios blocos. Blocos básicos [1], são porções de código que têm como *entry point* a primeira instrução do programa ou o alvo de um desvio, têm todas as suas instruções executadas em seqüência sem nenhum desvio até última instrução. Não é possível saltar para ou a partir de uma instrução no meio de um bloco básico. Paralelamente à divisão do código em blocos básicos, o algoritmo cria estruturas de controle auxiliares. São basicamente duas tabelas *hash*. Uma delas é usada para manter a relação entre o número do bloco e sua referência em memória. A outra, tem por objetivo manter uma relação entre o endereço da primeira instrução (no método em byte-code) de um bloco e o número do bloco. Essas estruturas são utilizadas em operações como adição, concatenação ou remoção de blocos básicos, auxiliando na manutenção correta dos ponteiros de sucessores e predecessores. O algoritmo cria em seguida um número de estruturas auxiliares que são úteis na coleta de informações sobre conjuntos de blocos básicos que formam sentenças de decisão (ex: IF's e WHILE/DO-WHILE's) e laços. Estas estruturas serão utilizadas no processo de reconstrução das ASA. Nas seções a seguir nós descrevemos este processo, mostrando como cada tipo de sentença é traduzida para JAS e quais são suas respectivas implicações.

O algoritmo 2 mostra o funcionamento do algoritmo de detecção de blocos básicos. Dado o código de um método (`ca`), este algoritmo constrói em `bblocks` (linha 4) uma lista de trechos de código correspondentes aos blocos básicos deste método. Para tanto, ele realiza duas passadas no método. Na primeira, compreendida entre as linhas 6 e

17, as instruções em byte-code são lidas do trecho de código (ca) e transformadas numa representação interna (linha 6). Além disso, são identificados os rótulos, alvos de desvios condicionais, condicionais compostos e incondicionais. Esses rótulos são armazenados temporariamente num conjunto de inteiros (`labelSet`) que se encontra vazio inicialmente (linha 2). Também nesta fase, as instruções lidas na linha 6 são armazenadas na lista encadeada `expList` (linhas 3 e 16). Finalmente, na segunda passada, utilizando os rótulos colhidos na primeira passada em `labelSet`, o algoritmo constrói a lista de blocos básicos em `bblocks` (linhas 20–24). Para isso, as instruções em byte-code vão sendo armazenadas em um bloco básico (linha 21) até o momento em que detecta-se que a instrução em análise se trata de um alvo de desvio ou de uma instrução de desvio (linha 22), caso em que o bloco atual é terminado e um novo bloco básico iniciado (linha 23).

Algoritmo 2 Algoritmo de detecção de blocos básicos

```
1   void parse(Code ca) {
2       IntSet labelSet = ∅;
3       ExpList expList = null;
4       BBList bblocks = null;
5       // Primeiro passada - Identificação dos labels
6       Exp exp = pickOpcode();
7       while(exp != null) {
8           if (isJump(exp)) {
9               // Salva alvo da instrução de desvio
10              labelSet.insert(exp.address + exp.getOffset);
11          } else if (exp instanceof of Switch) {
12              // Salva todos os alvos da instrução de desvio composto
13              labelSet.insert(exp.target());
14          }
15          // Acrescenta expressão na lista de expressões
16          expList.pushBack(exp);
17      }
18
19      // Segunda passada - Construção dos blocos basicos
20      while(expList != null) {
21          bblocks.addInstruction(expList.head);
22          if (isJumpTarget(expList.head.address) || isJump(expList.head))
23              bblocks.beginNextBlock();
24      }
25  }
```

Na tabela 4.2 nós apresentamos a descrição dos campos das estruturas auxiliares construídas para manter registrados os dados sobre início, fim, endereço da expressão de teste e do tipo de estruturas de alto nível como blocos if-then-else, laços while/do-while e blocos de tratamento de exceção try-catch-finally. Com esses dados em mãos, a detecção de tais estruturas é facilitada nas próximas etapas.

Campo	Descrição
start	Endereço de início da estrutura
end	Endereço do final da estrutura
test	Endereço da expressão de teste da estrutura (WHILE/DO-WHILE)
type	Tipo de estrutura

Tabela 4.2: Campos das estruturas auxiliares usadas na decompilação do byte-code Java

4.2.1 Sentença de Atribuição

Uma senteça de atribuição simples como `a = b + c` é transformada numa série de instruções JAS, como apresentado na figura 4.8. A tradução é feita de forma praticamente direta.

Linha	Java	Byte-Code	JAS
1			`STORE(0,`
2	`int a, b, c;`	`iload_1`	`  ADD(`
3	`a = b + c;`	`iload_2`	`  LOAD(1),`
4		`iadd`	`  LOAD(2)`
5		`istore_0`	`  )`
6			`)`

Figura 4.8: Exemplo de sentença de atribuição em JAS

Na figura podemos ver que a sentença `a = b + c` é traduzida para a seqüência de instruções em byte-code que carregam os operandos na pilha (linhas 2 e 3), somam os dois operandos, colocando o resultado de volta na pilha de operandos (linha 4), e finalmente salvam o resultado na variável `a` (variável 0 em byte-code). Em JAS, o funcionamento é similar ao do byte-code, apenas aproximando mais sua representação do código fonte. Os operandos são carregados e somados (linha 2, 3 e 4) e o resultado salvo na variável 0.

4.2.2 Sentença if-then-else

Basicamente, uma sentença if-then-else é traduzida na instrução CJUMP em JAS. A instrução CJUMP pode tomar dois ou três argumentos, dependendo da existência ou

não do bloco `else`: (a) uma expressão relacional; (b) as sentenças que são executadas caso a expressão relacional seja verdadeira; (c) as sentenças que são executadas caso a expressão relacional seja falsa. Para sentenças if-then-else, as estruturas auxiliares guardam o endereço de início do bloco do `then` e do bloco do `else`, o tipo do teste (ne, eq, lt, etc) e o endereço do fim do bloco. A figura 4.9 mostra uma sentença if-then-else em Java e sua tradução em byte-code.

Linha	Java	Byte-Code
1	`if (x == 0) {`	`iload_0 ; carrega x`
2	`y = z * 2;`	`ifne L1 ; salta para o ELSE se x != 0`
3	`z = 0;`	`iload_2 ; carrega z`
4	`} else {`	`bipush 2 ; carrega 2`
5	`y = x * 2;`	`imul ; z * 2`
6	`z = x;`	`istore_1 ; y = z * 2`
7	`}`	`iconst_0`
8		`istore_2 ; z = 0`
9		`goto L2 ; ignora o ELSE`
10		`L1: iload_0 ; carrega x`
11		`bipush 2 ; carrega 2`
12		`imul`
13		`istore_1 ; y = x * 2`
14		`iload_0`
15		`istore_2 ; z = x`
16		`L2: ...`

Figura 4.9: Exemplo de bloco IF-THEN

Do exemplo da figura, notamos que o teste da instrução `if` em byte-code possui o teste invertido em relação à mesma instrução em Java. Isso acontece porque a instrução `if` em byte-code nada pode fazer além de um desvio, caso a condição testada seja verdadeira. Caso o teste em byte-code seja falso (verdadeiro em Java), o processamento segue até a linha 9, executando a porção de código em byte-code correspondente ao bloco do `then`. Para evitar que também o código associado ao bloco do `else` seja executado, um *jump* é executado, saltando este trecho do código e continuando com a execução do programa. Caso o teste seja verdadeiro (falso em Java), a execução é desviada para o rótulo L1, executando assim o código em byte-code associado ao bloco do `else` (linhas 10–15).

O trecho de código da figura 4.9 aparece já traduzido para JAS na figura 4.10.

JAS

```
CJUMP(LOAD(0),
   // Bloco do THEN
   STORE(1,
      OPER(MUL,
         LOAD(2),
         CONST(2)
      )
   )
   STORE(2, CONST(0))
   // Bloco do ELSE
   STORE(1,
      OPER(MUL,
         LOAD(0),
         CONST(2)
      )
   )
   STORE(2, LOAD(0))
)
```

Figura 4.10: Trecho de código da figura 4.9 em JAS

4.2.3 Sentenças Encadeadas

Sentenças de controle como if-then-else encadeados são facilmente convertidos em JAS usando composição de instruções CJUMP. Por exemplo, a figure 4.11 mostra dois blocos `if-then`, as suas representações em byte-code, bem como suas respectivas traduções para JAS, utilizando CJUMP compostos.

O código Java apresentado na figura é um exemplo bastante simples de instruções `if` encadeadas. A interpretação deste trecho de código é bastante semelhante ao trecho do if-then-else apresentado anteriormente. Novamente aqui os teste das instruções `if` do byte-code são invertidos. Na linha 1 e 2 carregamos e testamos se `x` = 0 através do exame do valor da variável 1. Caso o valor de 1 seja diferente de zero (`ifne`), o fluxo de execução é desviado para o rótulo L1 (linha 16) e o corpo do primeiro `if` é saltado. Caso contrário (x = 0), o teste da linha 2 será falso e a execução prossegue na linha 3. Na linha 6 encontramos o `if` encadeado e o mesmo raciocínio se aplica. A única observação a ser feita no caso de `if's` encadeados é quando o início ou o final de um `if` mais interno coincide com o início ou final do `if` imediatamente mais externo. Neste caso os rótulos que marcam o final do bloco dos `if's` em questão podem coincidir. Um pequeno controle dentro do código de decompilação é necessário para identificar tais casos.

Código Java

```
if (x == 0) {
   w = 0;
   if (z == 0) {
      z = z * 2;
      w = 1;
   }
   y = z + w;
}
```

Linha	Byte-Code		Sintaxe Abstrata
1	iload_1	; loads x	CJUMP(x == 0,
2	ifne L1	; jumps if x != 0	STORE(w, CONST(0))
3	iconst_0	; loads constant 0	CJUMP(z == 0,
4	istore_0	; store 0 in w	STORE(z,
5	iload_3	; loads z	OPER(MUL, z, 2)
6	ifne L2	; jumps if z != 0	)
7	iload_3	; loads z	STORE(w, CONST(1))
8	iconst_2	; loads constant 2	)
9	imul	;	STORE(y,
10	istore_3	; z = z * 2	OPER(ADD, z, w)
11	iconst_1	; loads constant 1	)
12	istore_0	; w = 1	)
13	L2: iload_3	; loads z	
14	iload_0	; loads w	
15	istore_2	; y = z + w	
16	L1: ...		

Figura 4.11: Exemplo de blocos IF-THEN encadeados

4.2.4 Laços

A decompilação de laços de byte-code para JAS concentra-se basicamente na recuperação das sentenças WHILE e DO-WHILE, visto que a instrução de alto nível `for` em Java é traduzida para um laço WHILE em byte-code. Como discutido na seção 4.2, durante a detecção dos blocos básicos nós preenchemos algumas estruturas auxiliares com o endereço da primeira e a última instrução que compõe o laço, bem como o endereço da expressão de teste do laço.

A figura 4.12 mostra um exemplo de recuperação de um laço do tipo WHILE. Nesta figura nós mostramos um laço simples em que os números de 0 a 9 são impressos na saída padrão. Começando na linha 1, vemos que inicialmente temos a inicialização da variável responsável pelo controle do laço. Em byte-code, vemos que o fluxo de execução é desviado para o rótulo L2, onde fica exatamente o código responsável pelo teste da variável do laço. Esta abordagem é utilizada pelo compilador Java do *Java Development Kit* da Sun Microsystems. No entanto, podemos observar comportamentos diferentes na construção de laços por parte de outros compiladores. Em outras palavras, alguns compiladores Java podem vir a colocar o teste do laço no início do mesmo, ao invés de colocá-lo no final, como faz o compilador da Sun. Seguindo com a execução, caso o valor da variável esteja dentro dos limites impostos pela expressão de teste, o fluxo de execução é desviado para o rótulo L1, início do bloco do laço. A partir daí, após cada iteração, o valor da variável de teste do laço é testado e, quando a expressão de teste se tornar falsa, a execução continua na linha 11.

Linha	Java	Byte-Code
1	`i = 0;`	`iconst_0`
2	`while(i < 10) {`	`istore_1`
3	`  System.out.println(i);`	`goto L2`
4	`  i++;`	`L1: getstatic 10`
5	`}`	`iload_1`
6		`invokevirtual 19`
7		`iinc 1, 1`
8		`L2: iload_1`
9		`bipush 10`
10		`ifcmplt L1`
11		`...`

Figura 4.12: Exemplo de laço WHILE em byte-code

Na figura 4.13 nós podemos conferir como fica a tradução de um laço em JAS. Aqui aparecem algumas instruções em JAS ainda não vistas e que merecem alguns comentários. A instrução WHILE em JAS aceita dois argumentos: (a) o teste do laço, que normalmente

é uma expressão tipo OPER (operação de comparação, normalmente); (b) seqüência de instruções a serem executadas enquanto o teste resultar em valor verdadeiro. Na linha 2 podemos ver que a expressão de teste é `OPER(LT, LOAD(1), CONST(10))` correspondente às linhas 8–10 do trecho em byte-code da figura 4.12. Na linha 4 vemos a versão da instrução GET que é utilizada quando estamos diante da instrução em byte-code `getstatic` (linha 4 da figura 4.12) referenciando um atributo externo à classe. Caso o atributo pertencesse à classe em questão nós usaríamos a variante IGET da instrução. Por fim, na linha 7 da figura 4.13, nos vemos como a instrução de incremento de inteiros `iinc` em byte-code é traduzida para JAS.

Linha	JAS
1	`STORE(1, CONST(0))`
2	`WHILE(OPER(LT, LOAD(1), CONST(10)),`
3	`  INVOKE(VT, 19,`
4	`  GET(ST, 10),`
5	`  LOAD(1)`
6	`  )`
7	`  OPER(INC, 1, 1)`
8	`)`

Figura 4.13: Tradução para JAS do laço da figura 4.12

Também monitoramos saltos (*jumps*) condicionais e incondicionais existentes em laços que saltem para o teste condicional do laço ou para fora do mesmo. Estes casos são traduzidos respectivamente nas instruções CONTINUE e BREAK em JAS. Na figura 4.14 podemos ver um exemplo de laço DO-WHILE com a instrução `continue` em seu bloco. A descrição do funcionamento é bastante semelhante ao do exemplo do laço WHILE da figura 4.12 e será propositalmente omitido.

A figura 4.15 exibe um funcionamento bastante semelhante ao da figura 4.13. Porém, explicaremos aquelas instruções adicionais ainda não vistas. Basicamente, este é um laço do tipo DO-WHILE, o que explica a mudança para instrução DOWHILE do JAS na linha 2. Além disso, com a adição da instrução `if` que executa a instrução Java `continue` em caso verdadeiro, também tivemos que acrescentar as linhas 7 e 8 em JAS que corresponde ao trecho de código das linhas 6–9 em byte-code da figura 4.14.

4.2.5 Exceções

O tratamento de exceções em Java também tem sua instrução correspondente em JAS. Para isto é usada a instrução TRYCATCH em JAS que aceita dois argumentos: (a) bloco do `try` que será executado e monitorado para o caso de lançamento de exceção; (b) bloco

Linha	Java	Byte-Code
1	`i = 0;`	`iconst_0`
2	`do {`	`istore_1`
3	`System.out.println(i);`	`L1: getstatic 10`
4	`if (i % 3) continue;`	`iload_1`
5	`i++;`	`invokevirtual 19`
6	`} while(i < 10);`	`iload_1`
7		`iconst_3`
8		`irem`
9		`ifeq L2`
10		`iinc 1, 1`
11		`L2: iload_1`
12		`bipush 10`
13		`ifcmplt L1`
14		...

Figura 4.14: Exemplo de laço DO-WHILE em byte-code

do `catch` que é executado em caso de exceção. A figura 4.16 mostra um trecho de código em Java e seu equivalente em JAS. Não mostramos o trecho em byte-code pois o arquivo de classe Java possui uma tabela denominada Tabela de Exceções que contém os endereços do trechos de código a serem monitorados e a lista de exceções a serem monitoradas, bem como o endereço para onde se deve saltar na presença de uma exceção. Em outras palavras, as instruções de tratamento de exceção em Java não têm representantes em byte-code. Apesar deste fato, e através da consulta à Tabela de Exceções do arquivo de classe Java, nós fomos capazes de criar a versão JAS das instruções de tratamento de exceções.

4.2.6 Expressões de Teste

Um dos maiores problemas na recuperação de sentenças de controle, a partir do código objeto, é a recuperação de suas expressões de teste. Com o objetivo de acelerar a execução dos programas, as expressões de teste são normalmente compiladas para expressões de *curto circuito* [1]. Uma expressão de curto circuito é uma implementação de uma expressão de teste que usa identidades booleanas para agilizar a computação da expressão.

A figura 4.17 mostra a tradução de uma expressão de teste para uma expressão de curto circuito, onde o valor de `X.b` é apenas avaliado se `X.a` $\neq$ `true`. O valor de `X.a` é testado primeiro. Se `X.a` $\neq$ `0` (ie. `X.a` = `true`) o programa desvia para o rótulo `L1` (linha 3) para que seja executado o corpo da sentença *if* em Java. Neste caso, não é necessário testar o valor e `X.b`. Caso `X.a` = `false`, o valor de `X.b` precisa ser testado e,

Linha	JAS
1	STORE(1, CONST(0))
2	DOWHILE(OPER(LT, LOAD(1), CONST(10)),
3	INVOKE(VT, 19,
4	GET(ST, 10),
5	LOAD(1)
6	)
7	CJUMP(OPER(REM, LOAD(1), CONST(3)),
8	CONTINUE
9	)
10	OPER(INC, 1, 1)
11	)

Figura 4.15: Tradução para JAS do laço da figura 4.14

Linha	Java	JAS
1	A a = new A();	STORE(0, ALLOC(14))
2		TRY (
3	try {	INVOKE(VT, 32, LOAD(0))
4	a.run();	) CATCH (
5	} catch(Exception e) {	INVOKE(VT, 35, LOAD(0))
6	a.quit();	)
7	}	

Figura 4.16: Tradução de tratamento de exceções de Java para JAS

sendo `true`, `X.a || X.b` é verdadeiro, e o corpo do `if` é também executado. Caso ambos os valores `X.a` e `X.b` sejam falsos, o programa desvia para o rótulo `L2` (linha 6) e o corpo do `if` é saltado. A expressão de curto circuito resultante da expressão de teste (`X.a || X.b`) faz portanto uso da identidade booleana $true + x = true$, onde x é uma variável booleana arbitrária.

Apesar de agilizar a computação, o uso de expressões de curto-circuito tem um preço, que é a geração de fragmentos de código não estruturados [1]. Conforme discutido na seção 4.2.2 uma instrução CJUMP pode tomar dois operandos: (a) uma expressão de teste; (b) uma seqüência de instruções em byte-code formando o corpo a ser executado se o teste for verdadeiro. Por exemplo as instruções (1) e (2) formam o teste de um CJUMP que é avaliado pela instrução (3) `ifeq L1`. Os operandos deste CJUMP seriam portanto formados pelas instruções (4)–(6), e o rótulo para o qual o fluxo será desviado caso o teste seja verdadeiro `L1` está na instrução (7). Note que as instruções (4) e (5) formam uma segunda expressão de teste que é avaliada por (6) `ifne L2`. Ou seja, temos uma expressão de teste dentro do primeiro CJUMP. Esta segunda expressão de teste define um segundo

Linha	Java	Byte-Code
1	`if (X.a || X.b) {`	`aload_0`
2	`System.out.println(''Ok'');`	`getfield a`
3	`}`	`ifne L1`
4		`aload_0`
5		`getfield b`
6		`ifeq L2`
7		`L1: getstatic 10`
8		`getfield 13`
9		`lcd 12`
10		`invokevirtual 19`
11		`L2: ...`

Figura 4.17: Tradução de expressões de curto circuito de Java para byte-code

CJUMP cujo corpo é formado pelas instruções (7)–(10). No caso do primeiro CJUMP avaliar verdadeiro, durante a execução da sentença (3), o desvio da execução seguirá para a instrução em `L1` que é a primeira instrução do corpo do segundo CJUMP. Assim sendo, temos dois blocos não encadeados formando um fragmento de código não estruturado. Infelizmente, na presença de fragmentos de código não estruturados torna-se impossível usar CJUMPs para decompilação.

A alternativa que nós adotamos, para recuperar a expressão de teste original, foi encapsular expressões de curto circuito em byte-code como operandos de uma instrução de curto circuito em JAS chamada SHORTC. Deste modo, podemos usar uma única sentença CJUMP que avalia o resultado da expressão SHORTC. As instruções SHORTC em JAS recuperam apenas um único operador booleano (o operador de mais alto nível) da expressão de teste original. Apesar disto resultar na impossibilidade de se recuperar outras expressões de teste mais complexas esta abordagem mostrou-se suficiente para o nosso objetivos. Além disso, teste de igualdade de uma constante com uma variável formam a maioria das expressões de teste típicas, e estas continuam sendo capturadas usando CJUMPs.

Como resultado, as instruções em byte-code de (1)–(2) e (4)–(5) da figura 4.17 são encapsuladas como operandos da instrução SHORTC, juntamente com o operador `OR`. A instrução SHORTC fica então:

`SHORTC(OR, [aload_0 getfield a], [aload_0 getfield b])` A seqüência de ins-

truções de carregamento de X.a ([aload_0 getfield a]) é traduzida para JAS usando a instrução IGET da tabela 4.1. A decompilação do trecho de código da figura 4.17 em código JAS pode ser vista na figura 4.18.

Linha	Java	JAS
1	`if (a || b) {`	`CJUMP(SHORTC(OR, IGET(a), IGET(b)),`
2	`  System.out.println(''Ok'');`	`  INVOKE(19,`
3	`}`	`  GETFIELD(13,`
4		`  GETSTATIC(10)),`
5		`  LCD(12)`
6		`  )`
6		`)`

Figura 4.18: Fragmento de código da figura 4.17 traduzido para JAS

No trecho da figura 4.18 nós traduzimos a expressão de curto circuito para a instrução JAS SHORTC. Na linha 1 a expressão de curto circuito `a || b` é traduzida na expressão `SHORTC(OR, IGET(a), IGET(b))` diretamente. A instrução SHORTC retorna então penas um valor que corresponde ao resultado da avaliação da expressão booleana. O restante do código em JAS é semelhante a trechos já comentados e sua explicação será omitida.

4.3 Compressão de Árvore em JAS

Depois que as expressões são definidas e o programa em byte-code é transformado em um programa JAS usando as instruções descritas na tabela 4.1, obtemos uma grande Árvore de Sintaxe Abstrata que é então comprimida utilizando-se o algoritmo LZW [25] via gzip.

A primeira etapa é então a codificação dessa ASA. Como dissemos no início deste capítulo, uma vez que o byte-code Java já possui uma representação bastante densa e comprimida, para torná-lo ainda menor é necessário que também mantenhamos nossa representação bem pequena. Para tanto, decidimos também pelo byte como sendo o tamanho do nosso símbolo. Desta forma, codificamos toda a ASA de forma que, no momento de sua leitura, podemos determinar corretamente onde começa e termina cada instrução ou bloco de instruções.

A codificação da ASA gera uma *stream* que serve de entrada para o algoritmo LZW do gzip usado na compressão. Os resultados desta compressão são mostrados a seguir.

4.4 Resultados Experimentais

Nós usamos classes do Benchmark JVM98 para realizar os testes. O Benchmark JVM98 é composto de 8 benchmarks que variam desde testes de conformidade da Máquina Virtual Java até decodificação de audio ISO MPEG Layer-3 (MP3) e compiladores. Nós utilizamos o JVM Client98 versão 1.03. Os benchmarks no JVM Client98 são representados abaixo com uma curta descrição e o respectivo número de classes que compõe cada um deles. Assim como nos testes realizados e descritos no capítulo 3, nestes testes também utilizamos uma máquina com processador Intel Pentium II de 416MHz e 128MB de memória RAM rodando o Slackware GNU/Linux versões 4.0 e 7.0.

- **_200_check** (17 classes).
  Realiza alguns testes para assegurar que a JVM instalada provê um ambiente adequado para programas Java.

- **_227_mtrt** (26 classes).
  Raytracer que trabalha numa cena desenhando um dinossauro.

- **_202_jess** (151 classes).
  Este é um Java Expert Shell System baseado no CLIPS expert shell system da NASA.

- **_201_compress** (12 classes).
  Método Lempel-Ziv modificado (LZW).

- **_209_db** (3 classes).
  Realiza múltiplas funções de banco de dados num banco de dados residente em memória.

- **_222_mpegaudio** (55 classes).
  Aplicação que descomprime arquivos de audio em conformidade com a especificação de audio ISO MPEG Layer-3.

- **_228_jack** (56 classes).
  Gerador de parser Java baseado no Purdue Compiler Construction Tool Set (PCCTS).

- **_213_javac** (176 classes).
  O compilador Java javac, parte do JDK 1.0.2.

O total de 499 classes refere-se apenas aos benchmarks. Porém, o JVM Client98 Suite é composto de classes auxiliares utilizadas para apresentar resultados, emitir relatórios e

outras tarefas. O total final é composto de 1148 classes. Na tabela 4.3 podemos ver uma tabela com os resultados obtidos com a compressão de um subconjunto das classes deste benchmark.

Arquivo de Classe	Código	Final	gzip	Ganho
_200_check/PepTest.class	5480	2381	2809	7,81%
_213_javac/Assembler.class	3749	1138	1826	18,35%
_228_jack/ParseGen.class	3431	1035	1193	4,61%
_228_jack/Jack_the_Parser_Generator_Internals.class	3383	1053	1309	7,57%
_213_javac/RuntimeConstants.class	2552	539	1259	28,21%
_222_mpegaudio/d.class	2078	870	1274	19,44%
_213_javac/Constants.class	1476	473	814	23,10%
_213_javac/Type.class	1238	534	740	16,64%
_200_check/FloatingPointCheck.class	1131	404	480	6,72%
_213_javac/ConditionalExpression.class	1077	397	543	13,56%
_213_javac/TryStatement.class	1052	602	699	9,22%
_213_javac/BinaryClass.class	1017	530	639	10,72%
_213_javac/FieldDefinition.class	811	392	483	11,22%
Averages	**2652**	**796**	**1082**	**10,78%**

Tabela 4.3: Resultados experimentais

Todas as classes do Benchmark passaram pelo mesmo processo. Primeiramente, os códigos foram removidos dos arquivos de classe. Em seguida, foram submetidos ao processo de decompilação descrito neste capítulo, foram codificados e compactados com o gzip. Nós utilizamos o gzip versão 1.2.4 (18 Aug 93) com a flag *best compression* ligada. Da tabela 4.3 podemos ver que a razão de compressão do nosso método é invariavelmente melhor que as do gzip. A melhoria nas razões de compressão varia de 5% a 23%, dependendo do programa, e sua média fica em cerca de 11%. Isto pode ser explicado pelo uso das técnicas de decompilação sugeridas neste capítulo.

Capítulo 5

Conclusão e Trabalhos Futuros

Neste trabalho nós mostramos um método de compressão de byte-code Java, baseado na recuperação das Árvores de Sintaxe Abstrata, capaz de atingir razões de compressão em média 11% melhores que o algoritmo LZW (ex. gzip). O algoritmo de compressão descrito neste trabalho, será incorporado a uma Máquina Virtual Java voltada para dispositivos portáteis/móveis que está sendo desenvolvida no IC-UNICAMP. O objetivo principal deste projeto é desenvolver uma máquina compacta e eficiente que possa minimizar o uso de memória.

Existem algumas formas de se estender este trabalho. Entre elas está a possibilidade de se comprimir várias classes em um único arquivo. Esta técnica já é empregada pela própria Sun Microsystems (utilizando apenas o algoritmo LZW), para comprimir as classes básicas que compõe a JVM (*archives* Java). Tencionamos utilizar esta estratégia para comprimir várias classes simultaneamente utilizando o algoritmo descrito neste trabalho. Uma vez que todas as classes estarão no mesmo arquivo, os símbolos comuns às classes podem vir a ser codificadas como um único símbolo, aumentando assim as razões de compressão. Uma outra extensão natural deste trabalho é a criação de *plug-ins* para navegadores Internet, como Netscape Navigator e Internet Explorer, que possam realizar a descompressão dos pacotes ou archives Java de forma transparente ao usuário.

No decorrer deste trabalho, desenvolvemos algumas técnicas de decompilação que poderão no futuro contribuir para o projeto de um decompilador Java eficiente. Como mencionado anteriormente, aqui apenas descrevemos as técnicas de compressão. Ainda que tenhamos tido o cuidado de criar uma codificação simples da JAS, permitindo sua fácil decodificacao, não nos foi possível implementar as rotinas de descompressão por problemas de tempo. Também por restrições de tempo, ficamos impossibilitados de utilizar as técnicas descritas para agrupamentos de arquivos de classe Java, como é feito atualmente nos *archives Java* ou *jar files*, como são comumente conhecidos. A idéia era implementar o mesmo conceito substituindo-se o algoritmo LZW pela JAS.

Este trabalho foi publicado nos anais do IV Simpósio Brasileiro de Linguagens de Programação (SBLP'2000), ocorrido em maio de 2000, sob a forma do artigo entitulado "Byte-Code Compression Using Abstract Syntax Tree Recovery".

Referências Bibliográficas

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley, Boston, 1988.

[2] Guido Araujo, Paulo Centoducatte, Rodolfo Azevedo, and Ricardo Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Transactions on VLSI Systems, Special Section on System-Level Synthesis and Design*, to appear, March 2000.

[3] Guido Araujo, Paulo Centoducatte, Mario Cortes, and Ricardo Pannain. Code compression based on operand factorization. In *Proceedings of MICRO-31: The 31st Annual International Symposium on Microarchitecture*, pages 194–201, December 1998.

[4] Benda S. Baker, Udi Manber, and Robert Muth. Compressing differences of executable code. In *Workshop on Compiler Support for System Software*, May 1999. Online: `ftp://ftp.cs.arizona.edu/people/muth/exediff.ps.gz`.

[5] Timothy C. Bell, Jhon G. Cleary, and Ian H. Witten. *Text Compression.* Advanced Reference Series. Prentice Hall, New Jersey, 1990.

[6] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):520–540, April 1986.

[7] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. Conf. on Programming Languages Design and Implementation*, May 1999.

[8] S. Debray, W. Evans, and Muth R. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, May 1999.

[9] Editorial. The Future of Computing. *The Economist*, pages 79–81, sep 1998.

[10] Peter Elias. Interval and recency rank source coding: Two on-line adaptive variable-length schemes. *IEEE Transactions of Information Theory*, IT–33(1), 1987.

[11] Jean Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *SIGPLAN Programming Languages Design and Implementation*, pages 358–365, June 1997.

[12] M. Franz. *Code-Generation On-The-Fly: A Key to Portable Software.* PhD thesis, Swiss Federal Institute of Technology, 1994.

[13] Michael Franz and Thomas Kistler. Slim binaries. *Communication of the ACM*, 40(12):87–94, December 1997.

[14] Michael Franz and Thomas Kistler. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming*, pages 21–34, February 1999.

[15] Christopher W. Fraser and Todd A. Proebsting. Custom instructions sets for code compression. October 1995. `http://www.cs.arizona.edu/people/todd/papers/pldi2.ps`.

[16] Michael R. Garey and David S. Johnson. *Computers And Intractability: A Guide to the Theory of NP-Completeness.* Bell Laboratories, 1979.

[17] Bluetooth Special Interest Group. Bluetooth. *http://www.bluetooth.com*, 1999.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.

[19] D. A. Huffman. A method for the construction of minimum–redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[20] Jini. *http://www.sun.com/jini and http://www.jini.org.*

[21] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proceedings of MICRO–30: The 30th Annual International Symposium on Microarchitecture*, pages 194–203, December 1997.

[22] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transaction on Information Theory*, IT–22(1):75–81, January 1976.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1997. Online at `http://java.sun.com/docs/books/vmspec/index.html`.

[24] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *ACM Conference on Principles of Programming Languages*, pages 322–332, January 1995.

[25] Welch. A technique for high-performance data compression. *IEE Computer*, 17(6):8–19, July 1984.