4
Avaliação do Código Gerado

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

4.1.
Descrição dos testes

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

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

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

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

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

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

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

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

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

---

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

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

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

Máquina Virtual	Listagem do Programa

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

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

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

Tabela 26: Partes que diferem entre as plataformas

Descrição	Listagem do Programa

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

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

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

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

Tabela 27: Partes comuns às plataformas

Nome	Descrição do Teste	Código Java

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

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

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

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

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

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

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

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

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

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

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

4.2. Desempenho obtido nas plataformas

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

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

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

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

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

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

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

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

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

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

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

---

[12] YIHUEY (2004) conclui que nem sempre o JIT do J9 traz benefícios para o desempenho, pois existe um acréscimo de processamento que não pode ser desprezado.

[13] Vale lembrar que este foi um dos argumentos usados para justificar a criação da nova especificação.

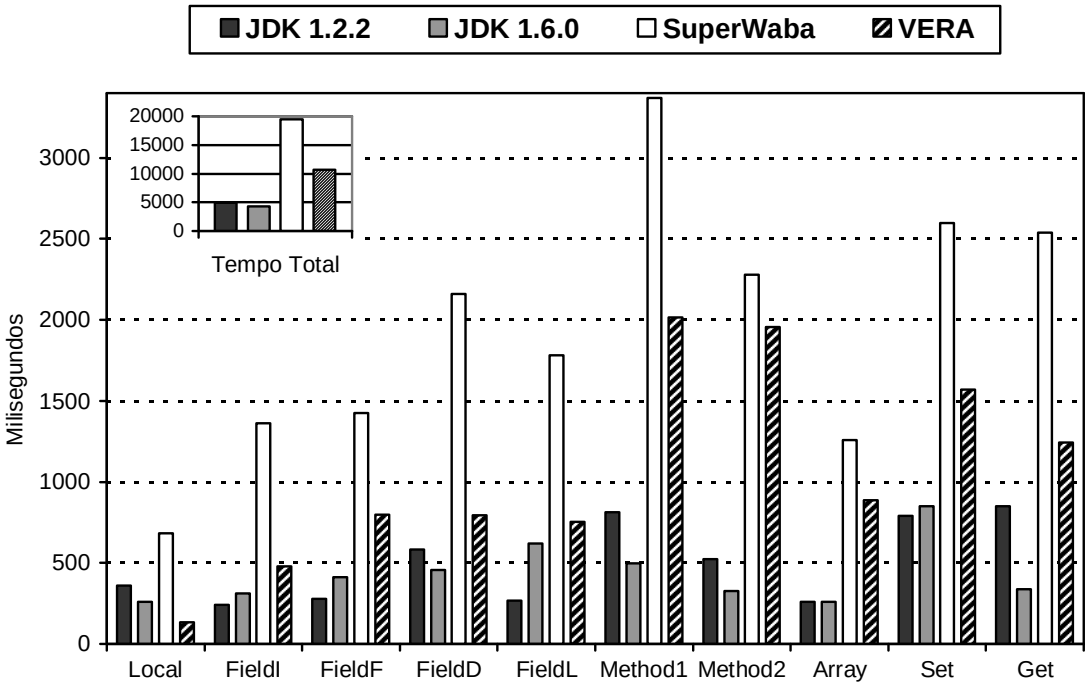

Figura 5: Gráfico comparativo com os tempos obtidos no Pentium 4-M

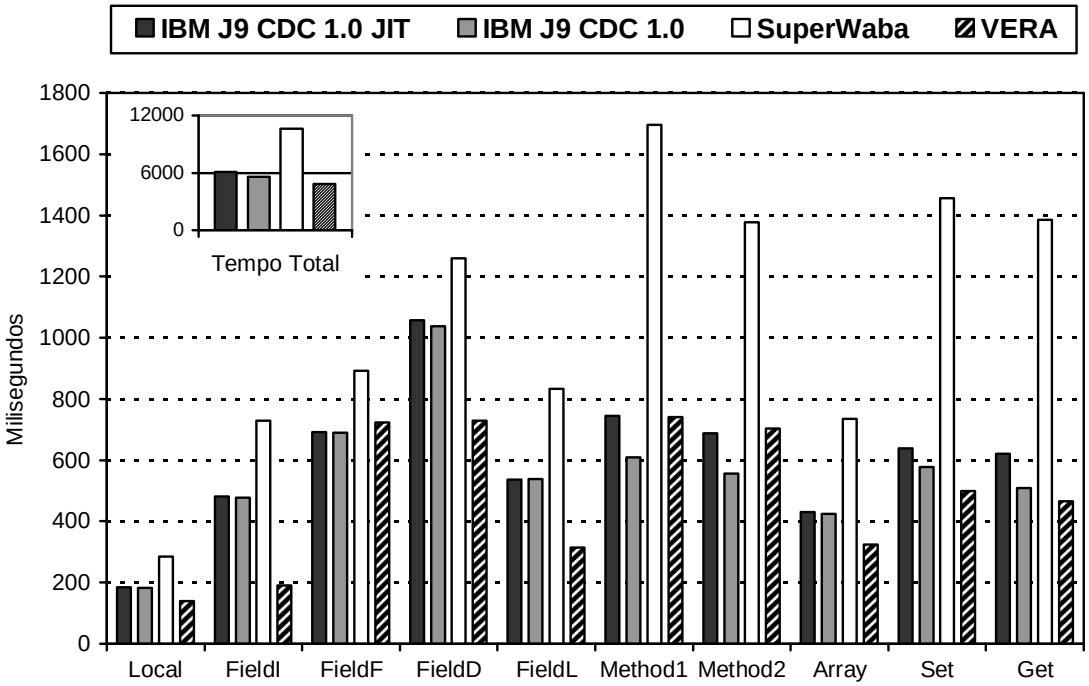

Figura 6: Gráfico comparativo com os tempos obtidos no Dell Axim X3

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

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

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

Tabela 30: Número de instruções dentro do laço

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

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

4.3. Tamanho do código gerado

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

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

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

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

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

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

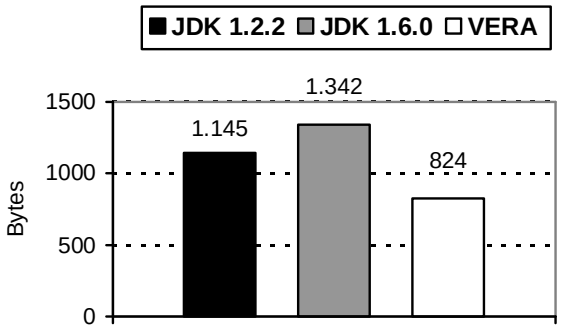

Figura 7: Tamanho do código gerado pela compilação do teste de desempenho
Foram removidos o construtor e os métodos `getTimeStamp` e `print`.

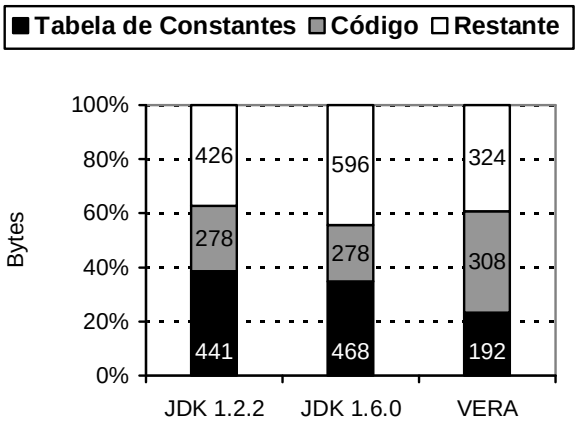

Figura 8: Composição do arquivo gerado pela compilação do teste de desempenho
Foram removidos o construtor e os métodos `getTimeStamp` e `print`.

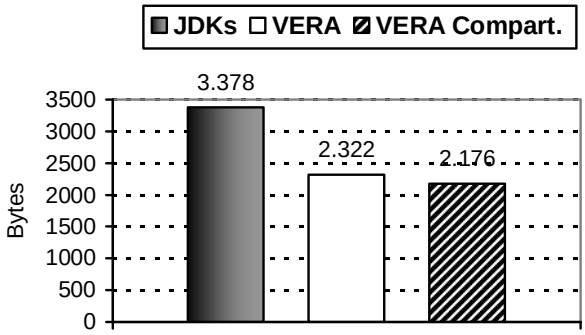

Figura 9: Tamanho do código gerado pela compilação dos protótipos