Generating Efficient Stack Code for Java

Tatiana Shpeisman and Mustafa Tikir
Department of Computer Science
University of Maryland
{murka,tikir}@cs.umd.edu

October 8, 1999

Abstract

Optimizing Java byte code is complicated by the fact that it uses a stack-based execution model. Changing the intermediate representation from the stack-based to the register-based one brings the problem of Java byte code optimizations into well-studied domain of compiler optimizations for register-based codes. In this paper we describe the technique to convert a register-based code into the Java byte code. The code generation techniques developed for the stack-based computers are not directly applicable to this problem as the comparative cost of the local memory and stack manipulation instructions in JVM is quite different from that in the stack-based computers. Naive verbose translation of the register-based code into the Java byte code produces the code with many redundant store and load instructions. The tool that we have developed allows to remove 90-100 % of the stores to the local (i.e., non-global) variables. It produces the Java byte code that is slightly faster and shorter than the original byte code even when no optimizations except for register allocation are performed on the register-based code.

1 Introduction

The Java programming language is becoming more and more popular as its design answers a demand for a completely-specified, portable and secure programming language. The only major complaint about Java is its performance. Optimizing Java is a wide-spread topic of the ongoing research. Developing the optimizations for the Java byte code rather than for the Java source code has several advantages: the byte code is independent from any compiler that was used to generate it, the byte code may be generated for languages other than Java and, finally, the byte code optimizations can be performed as a prepass to Just-In-Time (JIT) compilation.

Optimizing the Java byte code is complicated by the fact that it uses a stack-based execution model. The problems of performing static analysis and transformations on the stack-based code are well described by Raja Vallee-Rai and Laurie Hendren [VRH98]. They propose a Java byte code optimization framework called Soot that uses a register-based 3-addressed intermediate representation. Such a framework brings a problem of optimizing a Java byte code into well-studied domain of compiler optimizations for register-based codes. Yet, using the register-based representation for the Java byte code creates two additional problems: converting the byte code into a register-based code, and converting a register code into a stack-based Java byte code. While the first problem has been practically solved, much room remains in developing the techniques to convert the register code into an efficient byte code. The code currently generated by Soot is verbose and inefficient. In fact, transforming the byte code into a Jimple representation used by Soot and back into the byte code may increase the number of byte code instructions by about 50 %. This code size increase is due to the redundant store and load instructions that are introduced by verbose translation of the register code instructions into the stack code instructions.

There can be two approaches to generating the byte code that contains a reasonable number of local memory instructions: to generate the optimal byte code directly from the register-based code, or to, first, convert a register-based code into an inefficient byte code, and then optimize the byte code using the results of the analysis performed on the register code. We have developed the technique that is based on the second approach, that is, we first convert a register code into an inefficient stack code, and then optimize it using the results of live variable analysis performed on the register code. Our goal is to eliminate as many store instructions as possible without introducing additional instructions whose only purpose is to change the order of the values on the stack. (We allow the dup instructions to be used instead of loads when such a transformation is likely to enable a store instruction elimination.)

We believe that our goal is well justified. Although Java does provide the byte code instructions whose combination can be used to arbitrary reorder top four stack words, it is unclear whether using such instructions is better than using local memory instructions. Usually, the Java byte code is not executed by the stack CPU, but is interpreted or JIT compiled. The relative cost of the local memory and stack manipulation instructions depends on the implementation of JVM and/or JIT compiler and is not known in general. On the other hand, eliminating the store instructions without introducing the stack manipulation instructions should be always profitable. A byte code interpreter gets less instructions to interpret and perform. A JIT compiler gets less instructions to analyze, and more variable whose life range can be found without performing expensive global live variable analysis.

There has been some earlier work on generating an efficient stack-based code, but none of it can be directly applied to the problem of generating the stack-based code for Java. An early work by Bruno and Lassagne solved the problem of generating an optimal code that evaluates an expression without dependencies on the stack with a finite depth. Later work has concentrated on performing the peephole optimizations [Han89, Hay86]. Relatively recently Koopman has investigated the problem of eliminating memory instructions for a basic block or even the whole program [PJK94]. This work has provided us with useful insights on the problem of stack-based code generation. Yet, technique developed by Koopman is based on the assumption that a single local variable instruction is more expensive than a sequence of instructions that operate on the stack, and, thus, cannot be directly applied to the Java byte code.

The rest of the paper is organized as follows. Section 2 briefly describes our implementation of a Java byte code optimization framework that uses a register-based intermediate representation. In Section 3 we give a detailed description of the technique to convert a register-based code into an efficient Java byte code. The experimental results are described in Section 4. We finish the paper by giving our conclusions and acknowledgments.

2 General Framework

A compiler that optimizes Java byte code while working with the register code representation should perform the following three steps: convert a byte code into a register code, optimize the register code and convert the register code into the byte code.

Our main interest is in the last step. Yet, it is impossible to convert a register code into the byte code unless we first obtain the register code (When we started this work the Soot framework has not yet been publicly available). A commonly accepted technique for converting a Java byte code into a register code is based on simulating the run-time stack. We used a simpler approach and, much to our surprise, obtained a good register code. The first stage of our translation is to convert each stack instruction into a register code instruction, with the fixed register numbers assigned to the stack locations and local variables. The next step is copy propagation and dead-code elimination. The register code obtained after these two steps still has some extra copy instructions that correspond to the store operations in the byte code. To get rid of them we use a optimization that we call a "backward copy propagation".

As it may be guessed from its name, the backward copy propagation propagates the copies backward. Given a copy instruction with source and destination registers, it replaces a previous definition of the source register with the definition of the destination register, swaps the source and destination of the copy instruction and moves it to the position immediately after the replaced definition. An example of converting a byte code into the register code is given in Figure 1.

The only optimization that we perform on the register code before converting it back into a byte code is register allocation. Our register allocation algorithm is based on the graph coloring techniques [Muc97]. We construct the webs, i.e., the collections of definition-use chains that share a common use, perform simple register coalescing, build an interference graph and color it based on the priorities given to the webs according to static reference count.

3 Converting a Register Code into a Byte Code

Converting a register code into an efficient stack code is not a trivial task. A direct translation, when each register instruction is translated into a sequence of the byte code instructions that load the arguments on the stack, perform the necessary operation and store the result into a local variable, results in a code

iload_1	S0=R1	–	–	–
iload_2	S1=R2	–	–	–
iadd	S0=S0+S1	S0=R1+R2	R2=R1+R2	R2=R1+R2
istore_2	R2=S0	R2=S0	S0=R2	–
iconst_5	S0=5	S0=5	S0=5	S0=R5
a) Byte code	b) After direct translation	c) Copy propagation & dead-code elimination	d) Backward copy propagation	e) Dead-code elimination

Figure 1: Backward copy propagation example

with many redundant store and load instructions. In this section we describe the techniques that allow us to eliminate most of them.

A store instruction removes a value from the top of the stack and places it into a register. The subsequent load instruction copies the value from the local variable back on the stack. Under certain conditions, it is possible to remove both store and load instructions, thus, letting the value reside on the stack rather than in a local variable between its definition and use. If a store instruction is used by multiple load instructions it may be necessary to first replace them by dup instructions.

We shall say that a store instruction is local if its variable is dead at the end of the basic block, and global otherwise. Eliminating the global store instructions is much more complicated problem than eliminating the local ones. Fortunately, most of the stores in a stack code naively generated from the register code are the local ones. We do not attempt to remove the global store instructions. Further on we shall always mean that the store instruction under investigation is dead at the end of the basic block.

3.1 Overview of the algorithm

To convert a register code into the byte code we first perform a naive translation and then optimize the generated byte code. For each basic block we perform the following steps:

- eliminate store instructions that are followed by a single load.
- recognize and replace the patterns of two consecutive loads that can be replaced by a `dup2` instruction (e.g., `iload_1`; `iload_2`; `iload_1`; `iload_2` can be replaced by `iload_1`; `iload_2`; `dup2`.)
- eliminate store instructions that are followed by multiple loads
- recognize and replace the patterns that can be replaced by `iinc` instructions.

Our experiments have shown that the above order of the optimization steps works best. Most of the store instructions are followed by a single load instruction. Eliminating such instructions is relatively easy and allows to significantly reduce the size of the code. In fact, in our implementation we perform this step while converting the register code into a stack code rather than as a separate pass. Replacing two loads by a single dup instruction is an enabling transformation that allows to eliminate more store instructions at the next step. The increment instructions are introduced last as they hide the store instruction and prevent them from being eliminated.

After processing all the basic blocks we perform one additional optimization on the whole control graph. If all the predecessors of a basic block end with exactly the same sequence of instructions, this sequence is moved to the beginning of the basic block. This transformation reduces the size of the byte code but does not change the number of instructions being executed. The most common situation when it is applicable arises from conditional assignments.

In the remainder of this paper we shall describe our techniques for eliminating a store followed by a single load and a store followed by multiple loads in more detail.

3.2 Eliminating a store followed by a single load

Consider a store/load pair separated by some instruction sequence. The interleaving instructions consume some words from the stack and produce some words on the stack. We shall say that an instruction sequence is stack independent if it neither consumes nor produces any stack words. Here, we mean the commulative effect of executing the instruction sequence rather than a sum of the words consumed and produced by single instructions. For example, instructions `istore_0; iload_0` consume one word from

```
public final class E1 {
int a[],b[];
  public void foo(int i) {
    b[i]=a[i];
  }
}
```

a) Source code

```
R2=R0.b
R3=R0.a
R4=R3[R1]
R2[R1]=R4
return
```

b) Register code

	Code	Stack		
1:	aload_0	this	.	.
2:	getfield <Field int b[]>	b	.	.
3:	**astore_2**	.	.	.
4:	aload_0	this	.	.
5:	getfield <Field int a[]>	a	.	.
6:	**astore_3**	.	.	.
7:	**aload_3**	a	.	.
8:	iload_1	a	i	.
9:	iaload	a[i]	.	.
10:	**istore_4**	.	.	.
11:	**aload_2**	b	.	.
12:	iload_1	b	i	.
13:	**iload_4**	b	i	a[i]
14:	iastore	.	.	.
15:	return	.	.	.

c) Byte code obtained by naive translation

1: aload_0
2: getfield <Field int b[]>
4: aload_0
5: getfield <Field int a[]>
8: iload_1
9: iaload
10: **istore_4**
12: iload_1
13: **iload_4**
14: iastore
15: return

d) Removed 2 store/load w/o instruction reordering

1: aload_0
2: getfield <Field int b[]>
12: iload_1
4: aload_0
5: getfield <Field int a[]>
8: iload_1
9: iaload
10: **istore_4**
13: **iload_4**
14: iastore
15: return

e) Reorder instructions

1: aload_0
2: getfield <Field int b[]>
12: iload_1
4: aload_0
5: getfield <Field int a[]>
8: iload_1
9: iaload
14: iastore
15: return

f) All stores are removed

Figure 2: Eliminating store instructions followed by a single load instruction

Instruction	Stack State		Number of words	
			consumed	produced
load_0	r0	.	0	1
load_2	r0	r2	0	1
putfield <Field int x>	.	.	2	0
load_2	r2	.	0	1

a) A code fragment with two load instructions

Instruction	Stack State		
load_0	r0	.	.
load_2	r0	r2	.
dup_x1	r2	r0	r2
putfield <Field int x>	r2	.	.

b) Second load is replaced by `dup_x1`

Figure 3: Finding the right dup instruction

the stack and produce one word on the stack, while instructions `iload_0; istore_0` neither produce nor consume any stack words. The pair of store and load instructions separated by some instructions can be safely eliminated if and only if the interleaving instructions are stack independent. This condition trivially holds when the interleaving instruction sequence is empty, i.e. store is immediately followed by the load.

What do we do if the interleaving instructions are not stack independent? One solution would be to handle some special cases by introducing the instructions that interchange the values on the stack. Yet, our goal is to eliminate the extra instructions, so we would like to avoid introducing the new ones. Instead, we attempt to interchange the instructions so that store and load are separated by the independent sequence. This transformation is possible if there exist an independent instruction sequence ending with the store under consideration and it is legal to interchange it with the interleaving instruction sequence.

Interchanging two sequences of instructions is legal only if it does not violate the data dependencies and does not contradict to the precise exception model, i.e., does not change the relative order of the instructions whose execution may result in an exception being thrown. Any instruction that accesses an array element or an object field may throw an exception. We interchange two instruction sequences only if no more than one of them contains an array access, a field accesses or a function call; there are no flow, anti or output dependencies between the local variables; and there are no monitor instructions.

An example that illustrates the above technique is shown in Figure 2. The Java source code and the register code for method `foo` are shown in Figures 2a and b. The byte code produced by the direct translation of the register code shown in Figure 2c contains three pairs of store/load instructions: 3 and 11, 6 and 7, 10 and 13. The first two pairs can be simply removed as they are separated by the independent instruction sequences. The resulting code is shown in Figure 2d. The instructions 10 and 13 are separated by `iload_1` instruction that produces one value on the stack. To enable store/load elimination we interchange this instruction with the independent sequence $\{4,5,8,9,10\}$. In the resulting code (see Figure 2e) the store is immediately followed by the load. The final byte code with all the store instructions eliminated is shown in Figure 2f.

3.3 Eliminating a store followed by multiple loads

The problem of eliminating a store instruction whose variable subsequently is used by several load instructions can be reduced to two smaller problems: replacing a load that is preceded by another load by a dup operation and eliminating a store followed by a single load.

Consider two load instruction separated by several other instructions. If the interleaving instruction sequence does not produce any words on the stack it is possible to eliminate the second load by introducing

```
final public class E2 {
  int n;
  final public void foo(int []a, int k) {
   a[k]=++n;
  }
}
```

a) Source code

```
R3 = R0.n
R4 = 1
R3 = R3 + R4
R0.n = R3
R1[R2] = R3
return
```

b) Register code

1:	aload_0
2:	getfield <Field int n>
3:	iconst_1
4:	iadd
5:	**istore_3**
6:	aload_0
7:	**iload_3**
8:	putfield <Field int n>
9:	aload_1
10:	iload_2
11:	**iload_3**
12:	iastore
13:	return

c) After stores with single loads have been eliminated

1:	aload_0
2:	getfield <Field int n>
3:	iconst_1
4:	iadd
5:	**istore_3**
9:	aload_1
10:	iload_2
6:	aload_0
7:	**iload_3**
8:	putfield <Field int n>
11:	**iload_3** // to be removed
12:	iastore
13:	return

d) Reorder instructions

1:	*aload_0*
2:	*getfield* <Field int n>
3:	*iconst_1*
4:	*iadd*
5:	*istore_3*
9:	**aload_1**
10:	**iload_2**
6:	**aload_0**
7:	iload_3
new:	dup_x1 // inserted
8:	putfield <Field int n>
12:	iastore
13:	return

e) Replaced `iload_3` by `dup_x1`

9:	**aload_1**
10:	**iload_2**
6:	**aload_0**
1:	*aload_0*
2:	*getfield* <Field int n>
3:	*iconst_1*
4:	*iadd*
new:	dup_x1
8:	putfield <Field int n>
12:	iastore
13:	return

f) Eliminated pair `istore_3` and `iload_3`

Figure 4: Eliminating store with multiple loads

Benchmark	Class	Ratio of the transformed to the original code size		
		Soot Framework, size in instr.	Our technique, size in instr.	Our technique, size in bytes
CaffeineMark	FloatAtom	1.64	0.99	0.98
	LogicAtom	1.44	0.97	0.95
	LoopAtom	1.40	0.98	0.98
	MethodAtom	1.36	0.94	0.96
	SieveAtom	1.32	0.96	0.97
	StringAtom	1.21	1.00	1.00
SciMark	FFT	1.50	0.96	0.88
	JBLAS	1.55	0.99	0.93
	JBLASopt	1.79	0.99	0.94
	Jacobi	1.67	0.95	0.85
	LU	1.58	0.99	0.91
	MonteCarlo	1.67	0.96	0.95
FhourStones 2.0	Game	1.68	0.97	0.94

Table 1: Effect of the representation changes on the byte code size

a new dup instruction right after the first load. The choice of the dup instruction (`dup`, `dup_x1`, `dup_x2`, `dup2`, `dup2_x1` or `dup2_x2`) depends on the number of words being loaded and the number of words consumed by the interleaving instructions. If the number of the consumed words is larger than 3 for single word loads or 4 for double word loads the transformation is not possible, as Java byte code does not have the required dup instructions. Choosing the correct dup instruction is illustrated by the example shown in Figure 3. Here, the interleaving sequence consist of a single `putfield`instruction that consumes two words from the stack. The required dup operation is `dup_x1` that copies the word on the top of the stack two words below, thus keeping the two top stack words intact.

When, the second load instruction cannot be eliminated we attempt to reorder the instructions to enable the elimination. We look for a sequence of instruction that includes the first load instruction and both consumes and produces the same number of words as being consumed by the interleaving sequence. If such a sequence exist we attempt to interchange it with the remainder of the interleaving sequence.

As an example consider the byte code shown in Figure 4c. Two occurrences of `load_3` instruction are separated by instructions 8,9 and 10. These interleaving instructions consume two words from the stack and produce two words on the stack. Thus, we need an instruction sequence containing instruction 7 that produces and consumes two words, that is, instructions $\{6,7,8\}$. The remainder of the interleaving sequence is instructions $\{9,10\}$. Interchanging these two sequences of instructions yields the code shown in Figure 4d. Now it is possible to remove the load instruction 11 by inserting a new `dup_x1` instruction. In the resulting code shown in Figure 4e there is just one pair of store/load instructions that is eliminated as have been described in the previous section. The final code with all the stores eliminated is shown in Figure 4f.

4 Experimental Results

Performing the optimizations on the register code for Java programs is practical only if it is possible to convert the register code into an efficient byte code. In particular, simply converting a byte code into a register code and back should not result in the byte code that is worse than the original code. Such a transformation may even improve the byte code, if the original byte code contained extra load and store operations. In this section we present the experimental results that show that our techniques allow to generate the byte code that is no worse or even better than the original byte code.

We have performed our experiments on the class files from the CaffeineMark 3.0, SciMark and FhourStones 2.0 benchmarks. As our current implementation does not support exceptions we skipped the classes whose methods throw or catch exceptions. The class files from CaffeineMark and FhourStones benchmarks have been compiled using Sun's JDK 1.2 `javac` compiler with `-O` option. The SciMark benchmark contains pre-compiled class files.

For each class file we performed the following operations: converted the byte code of all the class methods into a register code, performed the register allocation on the register code and converted the

Package	Class	Store instruction ratio,%		
		Local/Total	Eliminated/Total	Eliminated/Local
CaffeineMark	FloatAtom	92	90	98
	LogicAtom	12	12	100
	LoopAtom	82	80	97
	MethodAtom	74	74	100
	SieveAtom	78	78	100
	StringAtom	83	83	100
SciMark	FFT	75	68	90
	JBLAS	58	58	100
	JBLASopt	86	81	94
	Jacobi	79	79	100
	LU	71	68	96
	MonteCarlo	77	72	94
FhourStones	Game	93	89	95

Table 2: Relative number of the eliminated store instructions

register code back into a byte code. We have measured how this sequence of representation changes affects the size of the code. We have also run the same transformations using Soot version 1.beta.1 [VRH]. The current version of the Soot framework generates a verbose and inefficient stack code. We use this code to demonstrate that a naive direct translation of the register code into a byte code really yields a very inefficient code.

Table 1 shows the ratio of the transformed code size to the original code size. The code generated by Soot Framework is about 1.5 times larger than the original byte code. Our technique generates the code that is even slightly shorter than the original byte code. For most of the class files, we obtain a better reduction in the number of bytes than in the number of the instructions. This is mainly the result of performing the register allocation (the store and load instructions for the registers 0-3 take just one byte rather than two). Also, dup instructions take only one byte. Thus, replacing loads to the registers with numbers 4 and higher by dup instructions reduces the number of bytes without reducing the number of instructions.

We have also measured the performance of the benchmarks before and after transformations. We have used Sun's JVM1.2 with enabled JIT compilation. Our transformed code got 1% better score on the Embedded CaffeineMark, 4.5% better score on the SciMark and the same score on the FhourStones benchmark

We have investigated how close to the optimal one is the stack code that we generate. We have measured the total number of store instructions in the naive code that we obtain by direct translation of the register code. An optimal code would have no store instructions at all. Our technique attempts to eliminate only the local store instructions, i.e., the instructions that store to a variable, that is dead at the end of the basic block. Table 2 shows the percentage of local store among all store instructions in the naive code, the percentage of the store instructions that have been eliminated and the percentage of local store instructions that have been eliminated. Our technique works pretty well. It eliminates at least 90 % of the store instructions. For six class files out of the thirteen that we investigated all local store instructions are eliminated.

5 Conclusions

A register-based code is a better studied and more convenient representation for performing program analysis and optimizations than the stack-based Java byte code. This paper presents the technique to convert a register-based code into an efficient Java byte code. Our transformation tool successfully eliminates 90-100 % of the local store instructions. The byte code obtained by simply changing the program representation from the byte code into a register-based code and back achieves 4.5% better score on the SciMark benchmark than the original byte code. Thus, the presented technique can be used within Java byte code optimization framework that uses a register-based intermediate representation without fear that inefficient translation of the register-based code into a stack-code would nullify the effect of optimizations performed on the register-based code.

The presented store elimination algorithm is mostly independent from the register-based code representation. It performs the optimizations directly on the byte code but needs the results of a live variable analysis that in our implementation is performed on the register code. Thus, the store elimination algorithm presented in this paper can be used by any compiler that generates the Java byte code or even as a stand-alone pass on the Java byte code itself.

Our current approach is based on the assumption that eliminating store instructions is always profitable as long as no instructions whose only purpose is to change the order of values on the stack is introduced. We would like to investigate how valid this assumption is for different implementations of JIT compilers, and whether it is possible to further improve the generated byte code by using more exact instruction cost model.

Acknowledgments

This work has started as a class project for CMSC731/838P "Programming Language Implementation: Implementing Java" taught by Dr. William Pugh at the University of Maryland. We thank Dr. Pugh for giving a wonderful class and encouraging us to continue with our work. We also thank all other members of the class who provided a supportive environment and beta tested the earlier versions of our code.

References

[Han89] T. Hand. Performance of the harris rtx-2000 c compiler. In *Proc. of the 1989 Rochester Forth Conf.,*, pages 61–62, June 1989.

[Hay86] J. Hayes. An interpreter and object code optimizer for a 32 bit forth chip. In *1986 FORML Conf. Proc.*, pages 211–221, November 1986.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman publishers, 1997.

[PJK94] Jr. Philip J. Koopman. A preliminary exploration of optimized stack code generation. *Journal of Forth Applications and Research*, 6(3):241–251, October 1994.

[VRH] Raja Vallee-Rai and Laurie J. Hendren. Soot: a java bytecode analysis and transformation framework. http://www.sable.mcgill.ca/soot/.

[VRH98] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical Report 1998-4, McGill University, July 1998. Available as `http://www.sable.mcgill.ca/publications/sable-tr-1998-4.ps`.