Bytecode Optimization

Domagoj Babić, Zvonimir Rakamarić
FER - Faculty of Electrotechnics and Computing, Zagreb, Croatia
{dbabic, zrakamar}@rasip.fer.hr

Abstract. *Today, when Java is entering the embedded market it needs performance enhancements more than ever. Large-scale enterprise applications would also benefit from better code optimization techniques.*

In this paper we present the results of optimization with an optimizing framework we have developed. We measure the impact of different optimizations on run times in different execution environments and propose further enhancements.

Keywords. Bytecode optimization, Java, optimization framework

1. Introduction

Execution environments for Java can be implemented in a few different ways. Two basic variants are virtual machines [1,2] and Java processors [3].

Contemporary Java virtual machines use interpretation combined with Just In Time (JIT) compilation for the execution of Java bytecodes. JIT compilers have to compromise between the time spend for code optimizations and the time left for program execution. The Jalapeno VM [2] spends around 93 percent of execution time running application code. Such a high percentage means that little time is left for JIT compiler and code optimization.

The short time frame left for code optimization prohibits the implementation of expensive code analyses and optimizations in JIT virtual machines and motivates better compiler optimization techniques.

As Java is becoming more popular in embedded devices, where processors are usually far less powerful, only the cheapest and the most effective optimizations can be supported. Expensive optimizations can be implemented in a compiler or in a separated optimizing package. We have devised a framework for Java bytecode optimization that can be used as a part of compiler or as a separate application.

Results also apply to JIT optimizing compilers. The framework is completely written in Java and it takes compiled Java class files as its input. After reading all the classes, the framework recovers the high-level structure of the program from its bytecode representation.

An abstract representation of the program is further analyzed and transformed into a more appropriate form for optimization. Analyses and transformations can be combined in an arbitrary way as far as certain dependency constraints are satisfied. For example, *BuildStackMap* analysis has to precede the transformation to the register representation of bytecodes that is further used for different optimizations. All precedence constraints are checked inside the framework. New analyses and transformations can be added easily.

In our research we evaluate the effect of our optimizing package on the average run times of the chosen Java applications. We have chosen a subset of JavaGrande [9] large-scale, computationally intensive benchmarks for measuring the effect of our optimizing package. Although such benchmarks are not considered as typical Java applications, we foresee that Java will have to handle more multimedia and encrypted content that is computationally demanding. Such applications are also a more probable load for an average user than compilers and parser generators that are often used to illustrate Java performance enhancements. Computationally intensive benchmarks are not forgiving even the smallest imperfections in code optimization what makes them very good code quality indicators.

Because of a large number of loops in code a small change in code can result in large speedups or slowdowns.

The main contributions of this paper are:

- a proposal of an extensible and modular architecture for optimization frameworks
- an elaboration of the register-stack code transformation and the color-graph local variable allocation algorithm effects

2. Framework

2.1. Architecture

We have based our design on standard programming patterns [4]. The basic control class is implemented as a Mediator Pattern. All transformations and analyses are performed upon containers that can contain Java packages, classes or methods. UML diagram of the framework architecture is shown in Fig.1.

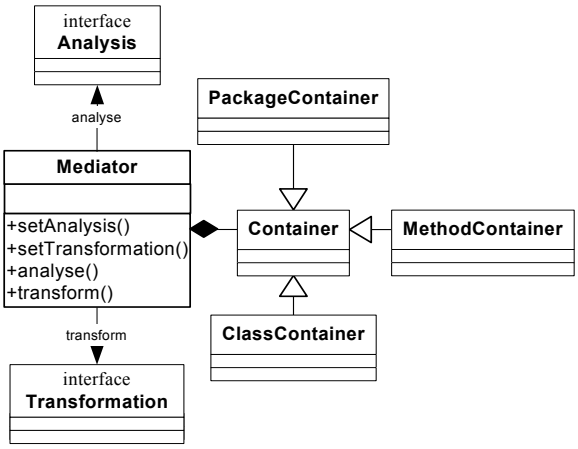

Figure 1. Framework UML diagram

All analyses, like control-flow and data-flow, subclass *Analysis* interface. A new current analysis can be set with a call to *setAnalysis* method with analysis type as a parameter. Subsequent calls to *analyse()* method with a container reference will append information produced by the current analysis to the container. Some analyses have order constraints that are checked before each analysis is performed. For example, data-flow analysis must precede color graph construction.

Classes that implement the *Transformation* interface make necessary updates of the program code and perform transformations between different code representations.

Relations between transformations and analyses form a complex directed graph. Since every transformation and analysis has to declare precedence constraints, the framework performs automatic verification of the optimization sequence.

To add a new transformation or analysis it is ample to subclass a suitable interface, declare precedence constraints and determine input and output representations. This way, almost all the effort can be directed to implementation of the needed functionality. This architecture will enable us to explore tangled Java optimizations faster and more efficiently.

Although some of frameworks mentioned in related work section describe basic framework architecture, according to our knowledge, none have addressed extensibility and modularity in-depth as we did. These features are extremely important if a large number of optimization possibilities are to be explored.

2.2. Containers

Containers can represent Java code in several different representations similar to representations in Soot framework [5]. When class files are read, the framework creates a bytecode level representation (BLR) for easier handling of packages, classes and methods.

BLR is an intermediate form that may also be used for disassembling class files.

Stack level representation (SLR) is used for control-flow and data-flow analysis and it is similar to Soot's Baf representation. We also use SLR for a graph-coloring analysis that reduces the number of used local variables and for peephole optimization.

To use standard code optimization techniques [6], stack code has to be transformed to three- address instruction form [that is called register level representation and is similar to Soot's Jimple]. Most of the optimizations are performed on the register level representation (RLR).

RLR is used as a basis for optimizations such as common sub-expression elimination, constant folding and dead code elimination.

3. Transformations and optimizations

According to Pugh [7], Java memory model prohibits some compiler optimizations. Pugh proposes a relaxed memory model that we use for our optimizations. An object field access or update cannot be moved across synchronization boundaries or synchronization points like running a new thread. We impose a strict order on write operations and permit read reordering if data dependencies allow it. Any method call between field referencing instructions might have a side effect, so we conservatively do not reorder such instructions.

We plan to implement side effect analysis in the future. Accepting these rules, the opti-

mizer cannot create invalid code for single processor machines, neither for properly synchronized multiprocessor programs.

Another problem in Java optimization are exceptions. Currently, all **athrow** instructions mark the end of the control flow block in our framework. An instruction that can throw an exception must not be moved outside of its catch block. More relaxed alternative to this rule is proposed in [8].

3.1. SLR to RLR

For transformation from SLR to RLR we use the stack map built by the dataflow analysis. Local variables can be mapped directly to three address instructions that are further used for code optimization. Temporary registers, used for storing values from the stack, are referenced by the slot index. Currently, the framework determines only the basic types of data in local variable and temporary registers. Example of SLR to RLR transformation:

```
SLR          RLR
iconst_1     #0int = 1
iconst_2     #1int = 2
iadd         #0int = #0int + #1int
store x      x = #0int
```

3.2. RLR to SLR

A sequence of bytecodes is defined for every register instruction in RLR. Example of RLR to SLR transformation:

```
RLR                      SLR
#0int = #0int + #1int    load #0int
x = #0int                load #1int
                         iadd
                         store #0int
                         load #0int
                         store x
```

It is obvious that such a transformation will generate load-store pairs that can be eliminated. We apply peephole optimization to such code to eliminate load-store pairs and to exchange expensive code patterns with cheaper ones.

Transformation from RLR to SLR leaves a messy code that has to be cleaned up using peephole optimization. At the moment, only basic pattern substitutions are performed. Some slow instruction sequences remain.

These patterns are easily optimized in JIT compiler and do not have an impact on JIT run-times. Interpreter, on the other way, cannot optimize them what results in slower execution times.

3.3. Optimizations

Implemented copy propagation optimization is local and propagates values only through basic control flow blocks. It scans all the instructions in the block and creates the list of *(variable, value)* pairs. If a *variable* from the list of pairs is later found on the right side of the association, it can be substituted with the *value*. The same optimization is applied to fields in objects.

The common sub-expression analysis scans bytecodes and creates *(expression, local variable)* pairs. If the same expression is later used, local variable can be used instead. For every instruction *x=expression1*, the pair list is searched for any expression equal to *expression1*. If there is such a pair *(loc_var_y, expression1)*, *x=expression1* is substituted with *x=local_var_y*.

Before running dead code elimination, we do the dataflow and the variable aliveness analysis on the code. If a local variable is not alive in the next instruction that means it is not used and we purge the instruction if it does not have any side effects. Our framework also performs constant folding.

The type analysis and the side-effect analysis [8] would enhance the efficiency of copy propagation and common sub-expression elimination, and we are currently implementing them.

Studying the results of the aliveness analysis, we have noticed that local variables can be allocated in less slots then javac (SUN JDK v1.4.0.) compiler does.

For that purpose we have devised a heuristic modification of graph-coloring algorithm that is similar to one presented in [12] with the difference that we do not reconnect node to graph once it is disconnected. Colors are allocated before disconnection.

Soot [5] framework uses interference graphs for this purpose and reports slowdowns caused by basic code transformations that include local variable packing and transformation from stack code to register code and back. They have not reported separate results for local variable packing and representation trans-

formations, as we will. According to our knowledge, other frameworks have not addressed these issues.

4. Performance evaluation

4.1. Benchmarks

For measuring the effect of our analyses and transformations, we have used benchmarks from JavaGrande benchmark suite. These benchmarks are computationally demanding Java applications that are available freely from the Internet [9]. We have chosen a subset of applications that have large processor power requirements, since our main intension is to cut on execution time applying code optimizations. All benchmarks we used are named in Table 1. Benchmark numbers from the table are used later as axis labels in speed-up bar graphs.

Table 1. Benchmarks

No	Benchmark	Code size [kb]
1	Crypt	8
2	FFT	8
3	LUFact	9
4	Series	26
5	SOR	14
6	SparseMatmult	6
7	RayTracer	6
8	Search	6

Crypt performs IDEA encryption and decryption of an array of 20M entries. FFT performs a one-dimensional forward transform of 8.4M complex numbers exercising complex arithmetic and trigonometric functions. LUFact is a Java version of popular Linpack benchmark that solves 1k X 1k linear system using LU factorization followed by a triangular solve. Series computes 100k fourier coefficients and exercises transcendental and trigonometric functions. SOR performs 100 iterations of successive over-relaxation on a 1500 X 1500 grid. SparseMatmult performs a 100k X 100k sparse matrix multiplication. RayTracer renders 3D scene containing 64 spheres at a resolution of 150 X 150 pixels. Search (also known as Fhourstones) solves positions in the game of connect-4 played on a 7x6 board.

The platform used for running benchmarks is Intel Klamath 333 MHz, 384 MB RAM and RH Linux v7.2, SUN Java v1.4.0 virtual machine. We use Linux *time* command to get the CPU time spent in benchmark execution in user mode.

We have created a TCL script that runs benchmarks, computes confidence intervals, compares results with Javac v.1.4.0 compiled code runtimes and generates a report. Script runs benchmarks 10 times and computes 95% confidence intervals. We could not measure any statistically significant difference between results obtained on lightly and heavily loaded machine. Anyway, we tried to keep the machine lightly loaded while benchmarks were running. Virtual machine start-up time is included in every execution time and for all benchmarks.

We decided to test our optimizations also with pure interpreted execution, because interpreters are better solution for some problems, for example where predictability is needed, and because we assume that the same optimizations that benefit interpreters would benefit direct Java execution on Java processors.

4.2. Results

The influence of basic SLR-RLR and RLR-SLR transformations is shown on Fig.2. Black bars represent JIT runtimes and hatched bars interpretation execution time. As described earlier, these transformations are necessary before any other optimizations can be performed and for creating stack code from register level representation. These transformations increase the number of unnecessary instructions that can be partially eliminated using peephole optimization. Another important effect is that local variables are allocated in ad-hoc manner and some slots might be empty.

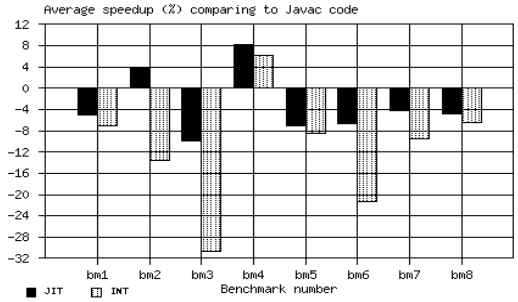

Figure 2. Basic transformations influence

The effect of remapping local variables with color-graph algorithm is shown in Fig. 3.

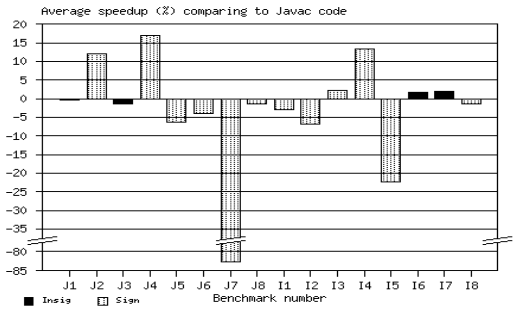

Figure 3. Graph-coloring effect

J4 means JIT execution speedup for benchmark number 4, the letter I is for interpreted execution. Black bar means that speedup is not statistically significant with 95% confidence interval.

Although some benchmarks perform better, it is surprisingly to see that more compact variable mapping can lead to large slowdowns, especially for JIT execution. Our heuristic graph-coloring algorithm achieved the average reduction of 0,44 local variable slots per method. Dense local variable mapping can result in more code register dependencies in JIT generated native code leading to less instruction level parallelism and slower code execution. The heuristic algorithm does not give special priorities to local variables in loops what can result in removing local variables used in loops in further slots. Currently we are working on a smarter local variable mapping.

In Fig.4. we present combined results from representation transformations and graph-color local variable mapping. One of negative effects, namely sparse local variable mapping, has been corrected with graph-coloring, so we get better runtimes than in Fig. 2, especially for JIT execution. Significant negative impact of graph coloring on RayTracer benchmark has remained.

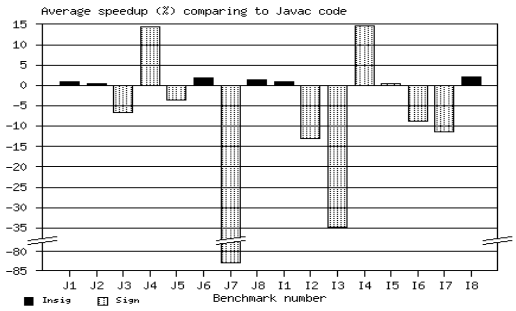

Figure 4. Combined effect

Run-times of benchmarks optimized with our framework without color-graph optimization are illustrated in Fig. 5.

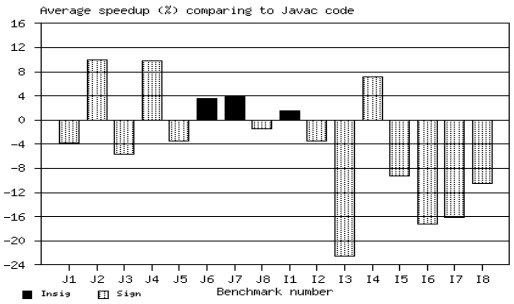

Figure 5. Optimized code speed-up

It's easy to see that optimizations have always a beneficial effect comparing it with artifacts created with basic necessary code transformations shown in Fig. 2. The effect of both optimizations and color-graph local variables mapping is illustrated in Fig. 6.

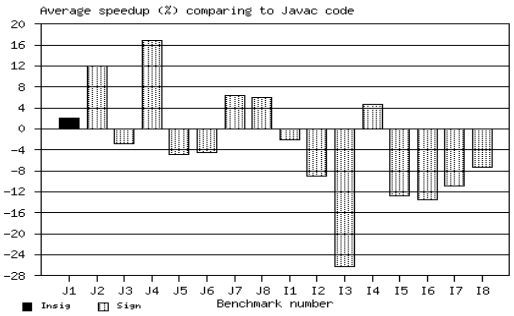

Figure 6. Optimized code with color-graph optimization

The average speed-up for JIT execution is 4.17% and speed-up computed as a weighted arithmetic mean is 13.81%. Only statistically significant values were included in computation. The weighted arithmetic mean is so large because Series benchmark represents 72.2% of total execution time for all benchmarks and has 16.89% speed-up. Such a mean tells us how much is the total run time for chosen benchmark kernel reduced, but 4.17% is a more reliable speed-up measure. It is interesting to note that the negative effect of graph coloring for the RayTracer (J7) benchmark is optimized away.

Interpreted execution times are slowed down 9.62% on average and weighted arithmetic mean is 5.55%. Although optimizations have ameliorated the negative impact of basic transformations (SLR2RLR and RLR2SLR), it is not enough to achieve the interpreted execution speed-up. The main reasons are code size

increase and inefficient code patterns resulting from basic transformations. Our average speed-up is comparable to results presented in [5], although we use different benchmarks. In interpreted execution our results are worse than in [5] because computationally intensive benchmarks are more sensitive to code imperfections.

5. Related work

There are a few Java optimization frameworks like Soot [10], Briki [11] and Cream [8]. Soot is the most advanced framework that includes the side effect analysis [13] and method inlining [14]. The architecture of Soot is not described in detail, but representations are explained. Briki optimizes array and field layout and does not implement any other optimizations. Cream has completely different architecture, without clearly defined representations and implements very detailed side effect analysis. All these frameworks work on compiled class files.

There are also other possibilities for Java optimization. A promising approach is attribute annotation to class files. These attributes give virtual machine more information about program. Significant speed-ups have been demonstrated in [15]. The major shortcoming of this approach is that most of virtual machines do not support these additional attributes.

Compiling Java to native code achieves the best performance but at the cost of portability [16].

6. Conclusion

A new architecture for optimization frameworks is proposed. The system is written completely in Java and it is easily extensible because of its modular architecture. We have described the design, functionality and efficiency of our framework.

We have analyzed the influence of basic transformations that are necessary for further optimizations and measured the influence of different ways to map local variables in slots.

Benchmarks we have used are computationally intensive and very sensitive to code quality. Our next goal is to improve the code quality of basic transformations and local variables mapping.

6. Literature

[1] Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley; 1999.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney. Adaptive Optimization in the Jalapeno JVM. 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization; 2000.

[3] U. Brinkschulte, C. Krakowski, J. Kreuzinger, Th. Ungerer. A multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling. IEEE PACT; 1999.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison-Wesley; 1995.

[5] Raja Vallee Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. Proceedings of CASCON '99; 1999.

[6] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley; 1985.

[7] William Pugh. Fixing the Java Memory Model. ACM Java Grande Conference; 1999.

[8] Lars R. Clausen. A Java Bytecode Optimizer Using Side-effect Analysis. Journal Concurrency: Practice and Experience; 1997.

[9] http://www.epcc.ed.ac.uk/javagrande/

[10] Raja Vallée-Rai. The Jimple Framework, Sable Technical Report, McGill University; 1998.

[11] Michal Cierniak, Wei Li. Optimizing Java Bytecodes. Concurrency: Practice and Experience; 1997.

[12] Dick Grune, Henri E. Bal. Modern Compiler Design. John Wiley; 2000.

[13] Chrislain Razafimahefa. A Study Of Side-Effect Analyses For Java. Thesis, McGill University; 1999.

[14] Vijay Sundaresan, Practical Techniques For Virtual Call Resolution in Java. Thesis, McGill University; 1999.

[15] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, Clark Verbrugge. A Framework for Optimizing Java Using Attributes. Computational Complexity; 2001.

[16] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, David Tarditi. Marmot: An Optimizing Compiler. Software – Practice and Experience, 1999.