Decompile for Visualization of Code Optimizations

Graduation Thesis

Erik Joelsson

TRITA-NA-Eyynn

Master’s Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology, April 2003
Supervisors at Nada was Mikael Goldmann and Viggo Kann
Supervisor at the project provider was Marcus Lagergren
Examiner was Stefan Arnborg
Abstract

The objective of this project was to recreate Java Source code from an intermediate code representation inside a modern JVM. The reason was to build a tool making it possible to visualize the result of optimizations.

The concept of backward transformations in code generation is not very well investigated compared to many other areas in computer science. Decompilers for Java byte code do exist, but none of them is doing a perfect job. This project was a bit different because it did not target Java byte code and the goal was not to produce legal Java source code but something more readable to humans than assembly language.

The focus of the project was on reconstructing higher levels of program structure, mainly by eliminating goto statements. Other areas such as type recollection was less touched upon.

Decomilation was, as expected, a rather difficult task. Even so, the project may be viewed as successful in producing a tool for optimization visualization.
Dekompilering för visualisering av kodoptimeringar

Examensarbete

Sammanfattning

Det här examensarbetet gick ut på att återskapa Java-källkod från en intermediär representation inuti en modern JVM. Målet med dekompileringen var att bygga ett verktyg för att visualisera resultatet av optimeringar.


Projektets fokus har legat på att återskapa programstrukturer genom att framförallt eliminera goto-satser. Andra områden, som till exempel att återskapa typinformation, har inte berörts lika mycket.

Dekompilering visade sig som väntat vara ett svårt problem, men arbetet får ändå ses som lyckat, då ett användbart verktyg för visualisering av optimeringar har konstruerats.
Acknowledgments

My greatest thanks go to Marcus Lagergren, my supervisor at BEA Systems. His enthusiastic commitment to the project has certainly been a welcome and sometimes needed fuel injection. I would also like to thank BEA Systems for giving me this opportunity even though times are hard in this business. Other people in BEA worth mentioning are Olof Lindholm and Joakim Dahlstedt who have both shown great interest in the project, and Björn Antonsson for reading and commenting on the report. Finally I want to give my thanks to my supervisors at Nada KTH, Mikael Goldmann and Viggo Kann.
Preface

Here follows an outline of this paper.

Chapter 1 gives an introduction to the project environment and defines the problem at hand. The intermediate code representation is presented.

Chapter 2 summarizes what previous work may be of help to the project and describes some theoretic concepts needed to understand the problem solution.

Chapter 3 describes the method used for solving the problem and why it was chosen. A detailed description on how the problem was solved is also given. The various phases of the decompilation are presented and all transformations used are described and exemplified.

Chapter 4 investigates some of the problems encountered during the project and gives suggestions on how to solve them.

Chapter 5 discusses the usefulness of the project and its result. A bit more thorough presentation of related work is also given.

Appendix A contains a list of the nodes used in the syntax tree representation in the decompilation process.
Contents

1 Introduction and Problem Definition .................................................. 1
  1.1 System Background ................................................................. 1
  1.1.1 BEA WebLogic JRockit ......................................................... 1
  1.1.2 Target Users ................................................................. 1
  1.2 Problem definition ................................................................. 1
  1.2.1 The Intermediate Code Representation .................................... 2
  1.2.2 Reconstruction versus Illustration ........................................ 2

2 Theory ......................................................................................... 4
  2.1 Related Work ............................................................................. 4
  2.2 Other Useful Theoretic Concepts ................................................ 5
  2.2.1 Syntax Tree ............................................................................. 5
  2.2.2 Program Point Analysis ......................................................... 5
  2.2.3 Dominator Relations and Loop Analysis .................................... 5

3 Method and Implementation ................................................................. 7
  3.1 Rebuild a Syntax Tree .................................................................. 7
  3.1.1 Language Extensions ............................................................... 7
  3.1.2 No Ramshaw ........................................................................... 7
  3.1.3 Set of Transformations ............................................................ 8
  3.2 Theoretic Concepts Applied ......................................................... 8
  3.3 The Four Phases of the Decompilation ............................................ 8
  3.4 Building the Syntax Tree .............................................................. 8
  3.4.1 Basic Blocks ........................................................................... 9
  3.4.2 Try-Catch ............................................................................... 9
  3.4.3 Loops ....................................................................................... 9
  3.4.4 Expressions ........................................................................... 10
  3.5 The Transformation Set .................................................................. 10
  3.5.1 Goto ....................................................................................... 10
  3.5.2 Basic Block ........................................................................... 12
  3.5.3 If and If-Else ......................................................................... 13
  3.5.4 While ...................................................................................... 14
  3.5.5 Do-While ................................................................................ 15
  3.5.6 Expressions ........................................................................... 16
  3.5.7 Switch ..................................................................................... 17
  3.5.8 Try-Catch ............................................................................... 17
  3.5.9 Lock and Unlock ..................................................................... 18
  3.5.10 Speculative Transformations .................................................. 19
  3.6 Goto Removal ............................................................................. 19
3.7 Clean Up ................................................................. 20

4 Results .................................................................. 21
  4.1 Decompilation problems ......................................... 21
    4.1.1 Continues in For Loops .................................... 21
    4.1.2 Assignment in Test ......................................... 21
    4.1.3 Variable Declarations ...................................... 22
    4.1.4 Generalized Properties and Transformations .......... 22
  4.2 Visibility of Optimizations ...................................... 23
    4.2.1 Local Variable Elimination ............................... 23
    4.2.2 Loop Restructuring ....................................... 23
    4.2.3 Speculative Inlining ...................................... 24

5 Conclusions and Recommendations ............................ 25
  5.1 Reactions from Target Users ................................... 25
  5.2 Conclusions ...................................................... 25

References ............................................................. 26

A Syntax Tree Nodes ................................................ 27
  A.1 Tree nodes ....................................................... 27
  A.2 Expression Tree Nodes ....................................... 28
Chapter 1

Introduction and Problem Definition

This chapter provides an introduction to the project including background knowledge and the main expected difficulties.

The reader of this paper is assumed to have a working knowledge of the Java™ programming language [7]. Other concepts, mostly regarding compilers and decompilation, will be described in as much detail as should be needed.

1.1 System Background

The project aim was to study and develop backward transformations in code generation in a modern Java Virtual Machine (JVM) [8].

1.1.1 BEA WebLogic JRockit

The company sponsoring the project was BEA Systems Inc. Among other things, BEA develops a high performance Java Virtual Machine (JVM) intended for the enterprise server market, called BEA WebLogic JRockit (from now on referred to as JRockit) [3]. The target usage has led to some important design choices. Instead of directly interpreting Java byte code, JRockit uses a technique called Just-in-Time (JIT) Compilation, meaning that a method is compiled just before its first use. This technique is combined with adaptive optimizations in order to increase JVM performance. The JIT compiler is simpler than the optimizing compiler to make startup time fast. Runtime profile data is used to find the most frequently executed methods which are then recompiled with various optimizations.

1.1.2 Target Users

There were two groups of users targeted by the project, both of them internal. One of them was the Java developers within BEA who might use this to learn how to write better Java code. The other was the developers of JRockit who wanted to see how well their optimizations work.

1.2 Problem definition

The goal of the project was to recreate something closely resembling Java source code from the JRockit intermediate code representation. The intermediate representation is derived from Java byte code. A program in Java byte code is not
guaranteed to have an equivalent in Java source code, since Java byte code is more expressive than Java source code. Optimizations on the intermediate representation also makes this more probable. The decompiler was therefore allowed to introduce some extensions to the Java language in the source code it produces.

The main expected difficulties were to reconstruct high level control structures from explicit jump instructions and to decipher the Java exception handling in try-catch blocks. Other tasks concerned recollecting type information and complex expressions.

### 1.2.1 The Intermediate Code Representation

The level of the intermediate representation was defined to be where the most optimizations are made and before it gets platform dependent. Variables are not yet register allocated and expressions are atomic assembly-like instructions.

The code is represented as a Control Flow Graph (CFG). The nodes of the CFG are the basic blocks of the code.

**Definition 1.1.** A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [2, p. 528]. In a JVM exceptions may always be thrown making it possible for a basic block to be interrupted.

The edges of the CFG correspond to the possible branches at the end of the basic blocks (See Figures 1.1 and 1.2).

### 1.2.2 Reconstruction versus Illustration

A balance will need to be struck between reconstructing correct Java source code and illustrating the structure of the compiled code. Taking this into account results in two aspects to be considered in the project. How accurate can Java source code be decompiled, and how accurate code is really needed. Ideally it should be possible for the user to decide how much the decompilation process is allowed to change the code structure.
Figure 1.2. This method contains a try-catch block. The dotted line symbolizes an exception handler relationship. In the Java source code block 1 is a try block and block 3 is its catch block.
Chapter 2

Theory

As stated in the previous chapter, there are several problems involved in decompilation. The main focus of this project was on reconstructing the higher levels of control flow in the Java language. This is also the area on which the most previous work has been focused.

2.1 Related Work

The problem of eliminating goto statements has been addressed before and Lyle Ramshaw has devised an algorithm for removing them without altering program structure [11]. He also showed that as long as a program does not contain global gotos or irreducible flow, such as loops with multiple entries, there exists a structural equivalent program without gotos.

Ramshaw’s algorithm has been successfully used by Proebsting and Watterson in Krakatoa, a Java byte code decompiler [10]. They found the technique successful in producing legal Java source code but they could not handle all the constructs of the Java language, most notably the Java exception handling.

Ramshaw’s algorithm works by creating a syntax tree by replacing all gotos and labels with labeled repeat loops, breaks and continues. These constructs are legal in Java but, according to many people, not very aesthetic. In Krakatoa a small set of transformations refines these further.

A rather innovative way of executing Java byte code was proposed by Ole Agesen [1]. He built a translator for Java byte code to Self byte code. His goal was to execute Java on a Self virtual machine since back then, the Self machine was more developed resulting in better performance.

Cifuentes has written a series of papers on decompilation [4, 5, 6]. Her projects have aimed at the more general case of real machine code decompiling with the goal of automatically translating programs from one machine architecture to another. Decompilation of actual machine code involves some major difficulties not found in this project, such as differentiating between code and data. The technique used to reconstruct high level control flow can be summarized as pattern recognition in a control flow graph.

For Java byte code there are a few commercial decompilers available. None of these works flawlessly. The most prominent is probably Jad [9] which is free for non-commercial use. Unfortunately it is not open source and not much information has been given out about how it works.
2.2 Other Useful Theoretic Concepts

Some theoretic concepts that were used in the project need to be introduced.

2.2.1 Syntax Tree

In compilers, a syntax tree is often used to represent the code in one of the intermediate representations. A syntax tree is basically a tree representation of the source code where nodes are operators and leaves are operands. A transformation from a syntax tree to legal source code may be viewed as a trivial task. More information about syntax trees can be found in any book on compiler construction, for example [2].

2.2.2 Program Point Analysis

A concept used in Krakatoa which was found very useful in this project is the program point [10].

Definition 2.1. A program point is a point before or after any program statement. A program point has two kinds of characteristics with regards to other program points, reachability and equality. A point B is reachable from another point A if there is a possible program flow from A to B. Two points are equivalent if every possible program flow from each of the points is the same.

2.2.3 Dominator Relations and Loop Analysis

Some standard concepts needed for the project are dominator relations and loop analysis in the CFG.

Definition 2.2. A basic block A dominates another basic block B if all control flows going through B pass through A.

Dominator information can be used to find natural loops.

Definition 2.3. A natural loop is defined by two conditions. The loop has a single entry point, called the header, which dominates all the basic blocks inside the loop. There must be at least one path from inside the loop back to the header.

To find a natural loop in a CFG search for an edge the head (target node) dominates the tail. Such an edge is called a back edge. Given a back edge A → B a natural loop is formed by B and the set of nodes that can reach A without going through B. B is the header of the loop. (See Figure 2.1) [2, pp. 602]
Figure 2.1. Example of a CFG containing a natural loop. Node 5 has a back edge to node 2 and node 2 dominates node 5. Nodes 3, 4 and 5 can all reach node 5 without going through node 2. The loop therefore consists of nodes 2, 3, 4 and 5 with node 2 as the loop header.

Figure 2.2. The dominance tree of the CFG in Figure 2.1.
Chapter 3

Method and Implementation

3.1 Rebuild a Syntax Tree

Any program written in Java source code can be structurally represented by a syntax tree. This is guaranteed by the high level control flow structures of the Java source code. Equipped with a good tree representation, the task of transforming it to legal Java source code is simple. The problem is then reduced to converting the CFG to a syntax tree for Java.

3.1.1 Language Extensions

A program in the intermediate code representation in JRockit is not guaranteed to be expressible by a syntax tree. Even if the original Java source code was well structured, the optimizations may very well change that. The correctness of this assumption regarding JRockit is one of the things this project may help gain some insight on.

Bad structure in this case means irreducible flow in the CFG. An example of irreducible flow is forward jumps into the body of loops. A program with irreducible flow cannot be expressed by high level control structures, as found in Java source code, it needs explicit jumps [2, pp. 606].

To be able to express every possible program in the intermediate representation some language extensions to the decompiled Java source code are needed. The first of these is naturally the goto statement with its corresponding labels.

The other extension regards synchronization. In Java, synchronization is achieved by a structured synchronized block. In Java byte code these are translated into simple monitor enter and monitor exit instructions [8], and in the intermediate representation, the are called lock and unlock. Due to irreducible or otherwise strange flow being possible, the complete recollection of these instructions to structured blocks is not guaranteed. Therefore lock and unlock statements were added to the decompiled Java source code.

3.1.2 No Ramshaw

Ramshaw’s algorithm for goto elimination was designed for reducible CFGs. Since it was assumed that the CFGs in this project may be irreducible Ramshaw’s algorithm was not used.
3.1.3 Set of Transformations

Accepting goto\textsuperscript{s} in the resulting code leads to the construction of a special kind of syntax tree containing goto\textsuperscript{s}. This will be referred to as a syntax tree for the remainder of this paper. The next step is to devise a set of transformations to be applied to the syntax tree repeatedly until no more changes can be done. These transformations will eliminate as many goto\textsuperscript{s} as possible and further structure the syntax tree. This idea resembles the method used in Krakatoa [10], except for not applying the transformation set on valid Java source code.

3.2 Theoretic Concepts Applied

The theoretic concepts described in the previous chapter were applied as follows.

In the syntax tree, a program point is a node associated with a before, after, first or last label. First and last take care of the case where a program point is located after the opening brace or before the closing brace of a block. In code examples in this paper, program points will be represented by $\phi_x$.

Loop analysis and dominator relations in the CFG were already implemented in the optimizer of JRockit, ready to be used.

3.3 The Four Phases of the Decompilation

The decompilation process is divided into four phases, each of which will be described in detail in the rest of this chapter.

Input to the process is the CFG of the intermediate code representation. The phases are:

1. Build a syntax tree from the CFG.
2. Apply transformations to gain a better structure.
3. Remove unnecessary goto\textsuperscript{s}.
4. Remove unnecessary labels and clean up empty if and else blocks.

The first step of the decompilation is building a syntax tree out of the CFG. In the initial step, the structure is not very tree-like, but through subsequent transformations applied to the tree, a better structure should emerge. When no more transformations can be applied, all unnecessary remaining goto\textsuperscript{s} are removed or converted to breaks or continues. Finally, empty if and else blocks and unnecessary block labels are cleaned up.

During the second phase, goto\textsuperscript{s} are seldom removed. Keeping goto\textsuperscript{s} around makes it possible to move around basic blocks quite freely, without making the program illegal. In the third phase, when the nodes of the syntax tree will not move around anymore, it is safe to remove ambiguous goto\textsuperscript{s}.

3.4 Building the Syntax Tree

When building the syntax tree, the basic blocks of the CFG are traversed in depth first order of the corresponding dominance tree but with a few exceptions explained below. A complete listing of the different kinds of nodes of the syntax tree can be found in Appendix A.
3.4.1 Basic Blocks

In the flow graph of the intermediate representation, basic blocks constitute a fundamental part of the representation and concept, but in the syntax tree they do not. However, completely discarding the grouping of operations the basic blocks provide would destroy some useful information, and therefore a remnant of this grouping is preserved. When the tree is created, basic blocks are translated to an abstract block node, providing a grouping of other nodes to make them easier to move around. This same block structure is also used by other typical block structures such as loops and if-blocks.

Basic blocks may be of one of five different kinds.

1. Fall through
2. 1-way branch
3. 2-way branch
4. n-way branch
5. Return

Fall through means that when execution reaches the end of the block it is continued at the beginning of the block situated directly after the current block. A 1-way branch block ends with an unconditional branch to another block. These two are considered structural equivalents and a goto node is always put at the end of these blocks.

A 2-way branch block must always end with some kind of compare instruction setting status flags. Based on the status, branches in two different directions are possible. A compare instruction is translated into an if-else construction with gotos for the two destinations.

The case with n-way branch blocks correspond to switch statements. Such a block ends with a switch instruction which branches according to a branch table. This is translated into a switch block with case blocks and a default block. The case and default blocks consist of only gotos.

For return blocks, no goto is appended.

3.4.2 Try-Catch

In the intermediate representation, try-catch blocks are represented by an abstraction called handlers. A catch block is a handler for every block in its corresponding try (see Figure 1.2).

A try block is created around the first occurrence of a block having a specific handler. By traversing the blocks in dominance order, the first block having a specific handler must be an entry block for this try block. Following every try block creation, catch blocks are created and placed together with their try block. All basic blocks having the same handler configuration will also be translated at this point and put inside the try block. All basic blocks will then be situated inside the correct try block from the start.

3.4.3 Loops

Natural loops are detected by loop analysis on the CFG. They may be of two kinds, while loops and do-while loops. The difference between them is that in a while
loop the loop condition is tested before the first iteration and in a **do-while** loop the first iteration is always executed. Both are initially constructed around just their header basic block, correctly nested in any **try** blocks, as infinite loops. The loop condition is a constant **true** statement. **For** loops are just another way of writing **while** loops and could be added later as syntactic sugar.

### 3.4.4 Expressions

Most instructions in the intermediate representation constitute some kind of expression. This includes for example arithmetic operations and method calls. Initially every such instruction is translated into one expression node, which in turn contains a tree structure describing the expression.

### 3.5 The Transformation Set

The following set of transformations are applied to the syntax tree to refine its structure. The tree is traversed over and over until no more transformations can take place. The idea behind these transformations is that it should always be good to apply a transformation when possible. They should not destroy any information or make other, possibly better choices of transformations, impossible to apply.

It is very important for this transformation set not to be infinite. With a few exceptions, every transform described here either reduces the size of the syntax tree, eliminates **goto**s or moves **goto**s out of block structures. No transformation introduces any **goto**s or moves them into blocks. The exceptions to this are the else creation transformation and the moving of basic blocks.

An **else** block is only introduced when there exist code that is only reachable by not entering an **if** block. On the other hand, an **else** block is only removed if it is empty, which means no code exist that is only reachable by not entering the **if** block. Thus the two rules will never be valid for the same **if-else** statement.

Basic block moving will never occur if the basic block is already positioned in the target position, or if another basic block is positioned there and it would be its target position. This prevents blocks from being moved in circles.

An important concept used when evaluating if a transformation is legal is the number of explicit and implicit predecessors of a basic block. Both of these are initialized with the number of **goto** nodes pointing to a block. They are then decreased by various transformations. The number of explicit predecessors indicates how many **goto** nodes there are left explicitly pointing at this basic block.

Every transformation below is presented along with an illustrated example, sorted according to the target node type.

#### 3.5.1 Goto

**Transformation 1 - Move Block to After Goto.** Moving basic blocks to after their only **goto** is one of the key transformations. Block moving easily corrupts the code structure and must therefore be heavily regulated. The number of implicit predecessor of the target basic block must be exactly one. The target basic block may not already be located after the **goto**. The starting basic block of a Java method may not be moved from its top position, since program execution in a method begins with the first statement. An ancestor of the **goto** node or a loop header for a loop
containing the \texttt{goto} node may not be moved to this \texttt{goto} node. The \texttt{goto} node and target basic block must share the same handlers. An exception to the last rule is if the target basic block is very small, only containing a \texttt{goto}, \texttt{return} or \texttt{throw} statement.

\begin{tabular}{|l|l|}
\hline
\textbf{Before} & \textbf{After} \\
\hline
\texttt{goto} block1; & \texttt{goto} block1; \\
\texttt{stmlist1} & \texttt{block1:} \\
block1: & (body of block1) \\
(body of block1) & \texttt{stmlist1} \\
\hline
\end{tabular}

Transformation 1: \textit{Move Block to After Goto}

\textbf{Transformation 2 - Convert Goto to Continue}. Converting \texttt{gotos} to \texttt{continues} is done to prevent back edges in loops from preventing block movement of the loop header. When a \texttt{goto} is converted to a \texttt{continue}, both the explicit and implicit number of predecessors of the target basic block are decreased.

To make the transformation legal the \texttt{goto} must be pointing at a header of a loop in which it is residing.

Labels are added if needed because of loop nesting.

\begin{tabular}{|l|l|}
\hline
\textbf{Before} & \textbf{After} \\
\hline
block3: & block3: \\
\textbf{while} (test1) \{ & \textbf{while} (test1) \{ \\
\texttt{stmlist1} & \texttt{stmlist1} \\
\texttt{goto} block3; & \texttt{continue} block3; \\
\texttt{stmlist2} & \texttt{stmlist2} \\
\} & \} \\
\hline
\end{tabular}

Transformation 2: \textit{Convert Goto to Continue}

\textbf{Transformation 3 - Convert Goto to Break}. Converting \texttt{gotos} to \texttt{breaks} is done to reduce the number of predecessors of the target basic block by combining a \texttt{goto} with another \texttt{goto} or \texttt{break}. When a \texttt{goto} is converted to a \texttt{break} both the explicit and implicit number of predecessors of the target basic block are decreased. Any kind of block, except basic blocks, may be subject to \texttt{break} statements.

To make the transformation legal there must be a \texttt{break} or \texttt{goto} directly after the block containing the \texttt{break} and the current \texttt{goto} must be pointing at the same block as this \texttt{break} or \texttt{goto}.

Labels are added if needed because of loop nesting or if a block type that is only breakable with labels is targeted.
<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>block3:</td>
<td>block3:</td>
</tr>
<tr>
<td><strong>while</strong> <em>(test1)</em> {</td>
<td><strong>while</strong> <em>(test1)</em> {</td>
</tr>
<tr>
<td><strong>goto</strong> block4;</td>
<td><strong>break</strong> block3;</td>
</tr>
<tr>
<td><strong>stmlist2</strong></td>
<td><strong>stmlist2</strong></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td><strong>goto</strong> block4;</td>
<td><strong>goto</strong> block4;</td>
</tr>
</tbody>
</table>

Transformation 3: *Convert Goto to Break*

### 3.5.2 Basic Block

**Transformation 4 - Move Basic Block to Optimal Position.** The optimal position for a basic block is a program point that is equivalent with the points after every predecessor to the basic block, and the control flow path from every such predecessor point passes this point. If such a program point exists for a given basic block, the basic block is moved to that point. To not cause an infinite transformation loop, a basic block is never moved to a point where a basic block is already positioned in its optimal position. This kind of block movement may also not violate any handler relations or loop memberships.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>block2:</td>
<td>block2:</td>
</tr>
<tr>
<td><strong>while</strong> <em>(test1)</em> {</td>
<td><strong>while</strong> <em>(test1)</em> {</td>
</tr>
<tr>
<td><strong>if</strong> <em>(test2)</em> {</td>
<td><strong>if</strong> <em>(test2)</em> {</td>
</tr>
<tr>
<td><strong>goto</strong> block3;</td>
<td><strong>goto</strong> block3;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td><strong>stmlist2</strong></td>
<td><strong>stmlist2</strong></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td><strong>block4</strong>:</td>
<td><strong>block3:</strong></td>
</tr>
<tr>
<td><strong>stmlist5</strong></td>
<td><strong>stmlist6</strong></td>
</tr>
<tr>
<td><strong>goto</strong> block5;</td>
<td><strong>goto</strong> block2;</td>
</tr>
<tr>
<td><strong>block3:</strong></td>
<td><strong>block4:</strong></td>
</tr>
<tr>
<td><strong>stmlist6</strong></td>
<td><strong>stmlist5</strong></td>
</tr>
<tr>
<td><strong>goto</strong> block2;</td>
<td><strong>goto</strong> block5;</td>
</tr>
</tbody>
</table>

Transformation 4: *Move Basic Block to Optimal Position*
3.5.3 If and If-Else

Transformation 5 - Move Common Gotos to After Else. If both an if block and its corresponding else block end with goto to the same target, one of the gotos is removed and the other is moved to after the else block.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (test1) {</td>
<td>if (test1) {</td>
</tr>
<tr>
<td>\hspace{1em} stmlist1</td>
<td>\hspace{1em} stmlist1</td>
</tr>
<tr>
<td>\hspace{1em} goto block3;</td>
<td>\hspace{1em} goto block3;</td>
</tr>
<tr>
<td>} else {</td>
<td>\hspace{1em} else {</td>
</tr>
<tr>
<td>\hspace{1em} stmlist2</td>
<td>\hspace{1em} stmlist2</td>
</tr>
<tr>
<td>\hspace{1em} goto block3;</td>
<td>\hspace{1em} goto block3;</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Transformation 5: Move Common Gotos to After Else

Transformation 6 - Create Short Circuit Or. Nested if statements as in the example are converted to a single if with short circuit or in the test expression. The program points $\phi_1$ and $\phi_2$ must be equal. This transformation has two symmetric cases and the other is obtained by inverting the inner if-else statement.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (test1) {</td>
<td>if (test1</td>
</tr>
<tr>
<td>\hspace{1em} $\phi_1$</td>
<td>\hspace{1em} goto block3;</td>
</tr>
<tr>
<td>\hspace{1em} goto block3;</td>
<td>\hspace{1em} }</td>
</tr>
<tr>
<td>} else {</td>
<td>\hspace{1em} else {</td>
</tr>
<tr>
<td>\hspace{1em} test2 {</td>
<td>\hspace{1em} stmlist1</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} $\phi_2$</td>
<td>\hspace{1em} }</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} goto block3;</td>
<td>\hspace{1em} }</td>
</tr>
<tr>
<td>\hspace{1em} else {</td>
<td>\hspace{1em} }</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} stmlist1</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} }</td>
<td></td>
</tr>
</tbody>
</table>

Transformation 6: Create Short Circuit Or

Transformation 7 - Create Short Circuit And. Nested if statements as in the example are converted to a single if with short circuit and in the test expression. The program points $\phi_1$ and $\phi_2$ must be equal. This transformation has two symmetric cases and the other is obtained by inverting the inner if-else statement.
Before After

```plaintext
if (test1) {
    if (test2) {
        stmlist1
    } else {
        φ2
        goto block3;
    }
} else {
    φ1
    goto block3;
}
```

Transformation 7: *Create Short Circuit And*

**Transformation 8 - Remove Empty If or Else.** If either of an if or an else block is empty it is removed. If an if block is removed the test expression is inverted and the contents of the else block is moved to the if block.

Before After

```plaintext
if (test1) {
    stmlist1
} else {
}
```

Transformation 8: *Remove Empty If or Else*

**Transformation 9 - Create Else.** If the basic block after an if block without an else is not reachable from any reachable point inside the if block, an else block is created around the basic block.

Before After

```plaintext
if (test1) {
    stmlist1
    goto block3;
} stmlist2
block3:
```

Transformation 9: *Create Else*

### 3.5.4 While

There are two transformation cases for while loops, each of which may occur in two symmetric variations.
Transformation 10 - Combine Infinite While and If. If a while loop is still infinite, the first statement in it is an if statement and the contents of either the if or else block is not part of the loop (by loop analysis), the if and while statements will be assimilated. The block that was not part of the loop will be put after the loop.

Before

```c
while (true) {
    if (test1) {
        goto block3;
    } else {
        stmlist1
    }
    stmlist2
}
```

After

```c
while (!test1) {
    stmlist1
    stmlist2
}
```

goto block3;

Transformation 10: Combine Infinite While and If

Transformation 11 - Combine While and If. If the first statement in a while loop is an if statement and the first program point in either of the if or else block is equal to the point after the loop, $\phi_1$ equals $\phi_2$ in the example, the if and while statement will be assimilated.

Before

```c
while (test1) {
    if (test2) {
        $\phi_1$
        goto block3;
    } else {
        stmlist1
    }
    stmlist2
} $\phi_2$
goto block3;
```

After

```c
while (test1 && !test2) {
    stmlist1
    stmlist2
}
```

goto block3;

Transformation 11: Combine While and If

3.5.5 Do-While

There are two transformations targeting do-while loops and one of them has two symmetric cases.

Transformation 12 - Combine Do and If. If a do-while is still infinite, the last statement in it is an if statement without else and the contents of the if block is not a part of the loop (by loop analysis), the test expression is inverted and set as the loop condition, and the if block content is put after the loop.
Before | After
---|---
do {
    stmlist1
    if (test1) {
        stmlist2
    }
} while (true);

Transformation 12: Combine Do and If

Transformation 13 - Combine Do and If-Else. If a do-while is still infinite, the last statement in it is an if-else statement, the first point in the if block is equal to the loop header and the contents of the else block is not part of the loop (by loop analysis), the else block content is moved to after the loop, and the test expression is inverted and set as the loop condition. A symmetric case is achieved by switching the if and else blocks.

Before | After
---|---
do {
    stmlist1
    if (test1) {
        continue
    } else {
        stmlist2
    }
} while (true);

Transformation 13: Combine Do and If-Else

3.5.6 Expressions

Transformation 14 - Combine Assignment with Expression. Expressions may be combined to build more complex expressions. An assignment expression is combined with an expression directly following the assignment if all of the following is true. The assigned variable must be local. The only occurrence of the variable, other than in the assignment, must be in the preceding expression.

Before | After
---|---
i3 = i2;
i4 = i3 + 6;
i4 = i2 + 6;

Transformation 14: Combine Assignment with Expression
### 3.5.7 Switch

**Transformation 15 - Reorder Cases.** This transformation orders case blocks to maximize fall through behavior. If the point before the last node of a case block is equal to the point at the beginning of another case block, the other case block is moved to right after the first case block and the last node of the first case block is removed.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>`switch (var) {</td>
<td>`switch (var) {</td>
</tr>
<tr>
<td>case 1:</td>
<td>case 2:</td>
</tr>
<tr>
<td>goto block2;</td>
<td>stmlist1</td>
</tr>
<tr>
<td>case 2:</td>
<td>case 1:</td>
</tr>
<tr>
<td>stmlist1</td>
<td>goto block2;</td>
</tr>
<tr>
<td>goto block2;</td>
<td>default:</td>
</tr>
<tr>
<td>default:</td>
<td>stmlist3</td>
</tr>
<tr>
<td>}</td>
<td>return;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

**Transformation 15: Reorder Cases**

**Transformation 16 - Convert Common Gotos to Breaks.** If all case blocks end with a common goto, except for those ending with either a return or throw statement, a goto to the same target is placed after the switch block and all the other gotos are replaced by breaks. To ensure this transformation is only applied once, no nodes may be residing directly after the switch block.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>`switch (var) {</td>
<td>`switch (var) {</td>
</tr>
<tr>
<td>case 1:</td>
<td>case 1:</td>
</tr>
<tr>
<td>stmlist1</td>
<td>stmlist1</td>
</tr>
<tr>
<td>goto block2;</td>
<td>break;</td>
</tr>
<tr>
<td>case 2:</td>
<td>case 2:</td>
</tr>
<tr>
<td>stmlist2</td>
<td>stmlist2</td>
</tr>
<tr>
<td>goto block2;</td>
<td>break;</td>
</tr>
<tr>
<td>default:</td>
<td>default:</td>
</tr>
<tr>
<td>stmlist3</td>
<td>stmlist3</td>
</tr>
<tr>
<td>return;</td>
<td>return;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>`goto block2;</td>
</tr>
</tbody>
</table>

**Transformation 16: Convert Common Gotos to Breaks**

### 3.5.8 Try-Catch

**Transformation 17 - Move Common Gotos to After Catch.** If a try block and all its catch blocks are all ending with a goto to the same target, except for blocks ending with return or throw, these gotos are replaced with one goto after the last catch block.
Transformation 17: Move Common Gotos to After Catch

3.5.9 Lock and Unlock

**Transformation 18 - Convert Lock and Unlock to Synchronized.** In Java, code synchronization is accomplished by `synchronized` blocks. Java byte code as well as the intermediate representation instead uses explicit `lock` and `unlock` instructions. A `try-catch` block is also created around the body of the `synchronized` block to make sure `unlock` is called even if an exception is thrown.

When a `lock` statement is encountered its corresponding `try` block should be located right after it. This `try` block is searched for all its exits to find all `unlock` statements belonging to this `lock` statement. If `unlock` statements are found for each exit of the `try` block, the `try-catch` is removed and the contents of the `try` block is moved into a newly created `synchronized` block. The `lock` statement and all previously found `unlock` statements are also removed.

**Transformation 18: Convert Lock and Unlock to Synchronized**

**Transformation 19 - Remove Synchronized Block from Synchronized Method.** If a method is synchronized and all the relevant code of the method is enclosed inside a `synchronized` block with this as synchronization object, the `synchronized` block is removed since it is redundant.
Before | After
--- | ---
public synchronized foo() {
  synchronized (this) {
    stmlist1
  }
} | public synchronized foo() {
  stmlist1
}

Transformation 19: *Remove Synchronized Block from Synchronized Method*

### 3.5.10 Speculative Transformations

To handle some really tricky situations, a second set of transformations was devised. When no more transformations are possible, unwanted *gotos* are checked for. Any *gotos* that would be handled by the *goto* removal phase are ignored. If there are *gotos* left a different set of transformations is applied but immediately terminated if one of them succeeds. The ordinary transformation set is then restarted.

The idea behind this set of transformations is that they may not always be good, but when faced with no other solution they should not cause more harm than good. Only one transformation was constructed for this set.

**Transformation 20 - Move Goto out of Loop.** If an infinite loop is found, not having an explicit jump after it and containing *gotos*, one of these *gotos* is moved to after the loop and replaced by a *break*.

Before | After
--- | ---
while (true) {
  v1 = v1 + 1;
  if (v1 > 5) {
    goto block3;
  }
  stmlist1
  if (v2 < 10) {
    goto block3;
  }
  stmlist2
} | while (true) {
  v1 = v1 + 1;
  if (v1 > 5) {
    break;
  }
  stmlist1
  if (v2 < 10) {
    goto block3;
  }
  stmlist2
} | goto block 3;

Transformation 20: *Move Goto out of Loop*

### 3.6 Goto Removal

The tree is traversed once, checking every *goto* statement for program point equality before and after the statement to remove unnecessary *gotos*. If a *goto* statement is located inside a loop or other non-basic block, equality is checked with the points before and after loops and after other blocks to possibly convert the *goto* into a *break* or *continue* statement.
This phase is separated from the second phase because removing \texttt{gotos} destroys information. At this stage though, that information is not needed anymore, since basic blocks will not be further moved around the tree.

3.7 Clean Up

After all possible \texttt{gotos} have been removed, the unneeded block labels are made invisible. Empty \texttt{if} or \texttt{else} blocks may also have emerged if they previously only consisted of a single \texttt{goto}. These empty blocks are also removed.
Chapter 4

Results

4.1 Decompilation problems

As was expected, a perfect decompilation process is very difficult, if not impossible, to achieve. The implementation presented here successfully decompiles a lot of code but there are still unsolved problems. Examples of problematic decompilations are presented in this chapter, along with discussions on how the problems might be solved.

4.1.1 Continues in For Loops

If a for loop has an increment block and any continue statements inside the loop will point to that block instead of the loop header. This is because the increment block is both the last statement of the loop and conceptually part of the loop header, but JRockit does not consider it to be part of the loop header. This makes it hard to detect the actual back edges and thus converting goto to continues. A good transformation for finding these increment blocks and combine them with the loop statement would be needed to solve this problem (See Figure 4.1). Also, JRockit could be modified to present loop information in a more flexible way when it comes to loop headers. Given more time, this problem would have been solved.

4.1.2 Assignment in Test

If an if statement in the original Java source contains an assignment in the test expression the assignment will not be put inside the test by the decompiler. Instead, the assignment will be left right before the if statement. This assignment will then effectively prevent any short circuit and or creation with an outer if statement, since the if statements would not be correctly nested.

Letting assignments go into test expressions would be the solution. The problem is obviously the risk of building too long and unnecessarily complex expressions. This kind of transformation would probably be best suited for the special transformation set.

If the assignment of v1 in Figure 4.2 was included in the test of the inner if statement, a short circuit and could be created. This would in turn lead to an else creation thus removing the need for the goto statement.
block2:
while (i < 100) {
    if (test1) {
        goto block3;
    }
    goto block3;
}
block3:
i = i + 1;
goto block2;

Figure 4.1. An example of a for loop with continue statements

if (test1) {
    v1 = v2.foo();
    if (v1 == 17) {
        goto block3;
    }
}  
block3:
i = i + 1;
goto block2;

Figure 4.2. An example of when an assignment in a test leaves a goto

4.1.3 Variable Declarations

Very little effort was put into recollecting type information. The scheme used was to just go through the syntax tree once and propagate local variable types in every assignment. When methods are printed, all local variable declarations are put in the beginning of the method.

If a local variable is actually used in several places with different scope and different types, the declaration in the beginning of the method may be incorrect. A more detailed type recollection would be needed (See Figure 4.3).

The decompiled code does not preserve the names of local variables. Usually local variable names are already lost in Java byte code, but when compiled with debugging information, variable naming is preserved. Such information was not used by the decompiler.

4.1.4 Generalized Properties and Transformations

The implementation of some concepts and analysis tools could have been more elaborate and generalized. Useful information could be reachability of a given point
Current Wanted

```java
Foo foo1;
if (test1) {
    foo1 = new Foo();
    foo1.doFoo();
} else {
    foo1 = new Bar();
    foo1.doBar();
}
```

```java
if (test1) {
    Foo foo1 = new Foo();
    foo1.doFoo();
} else {
    Bar bar2 = new Bar();
    bar2.doBar();
}
```

Figure 4.3. Example of when variable type declaration fails

from anywhere inside a given block, or to find all exits of a block. This would have made it possible to generalize some of the transformations to handle some rare but really tricky situations. Again, this kind of information is possible to obtain and would have been implemented given more time.

4.2 Visibility of Optimizations

The actual goal of the project was visualizing optimizations. Some kinds of optimizations show rather well in the decompiled source code, other kinds do not.

4.2.1 Local Variable Elimination

When expressions are combined, local variables may be eliminated. If a later optimization eliminates this variable it will not make any difference to the decompiler. Scheduling optimizations may also prevent some expression combining and falsely give the impression of more work being done after the optimizations.

4.2.2 Loop Restructuring

Some structural optimizations show rather well. Especially things like loop unrolling (See Figure 4.4). Other kinds, such as inverting of tests may well be changed again by the decompiler.

```java
Before After

i7 = 0;
while (i7 < i3) {
    i7 = i7 + 1;
}
```

```java
i25 = 0;
i24 = 0;
if (0 < i23) {
    do {
        i25 = i24 + 1;
        i24 = i25;
    } while (i24 < i23);
}
```

Figure 4.4. Example of loop restructuring
4.2.3 Speculative Inlining

In an object oriented environment like Java, inlining method calls can be very hard. The problem is the dynamic linking often making it impossible to know the exact type of an object before execution. To circumvent this, JRockit uses speculative inlining. This means that a method call gets inlined with a probable version of the method but with a type check on the object. If the type check fails the real call is made instead. These type checks are easily detected making this kind of optimization very obvious in the decompiled code.
Chapter 5

Conclusions and Recommendations

5.1 Reactions from Target Users

The decompiler has so far only been used by JRockit developers. The main usage has been as a debugging tool to inspect the state of the intermediate representation. This is especially useful for automatically generated classes where there is no source code or even a class file available.

Another practical usage is to turn on the decompiler and let it dump everything that gets compiled for a new Java application being tested. Optimizations may then easily be searched for and studied.

Of course there are some areas needing improvement. Some methods do not get as readable as they should. This was partly covered in the previous chapter.

Together with other tools for visualizing the CFG (as seen in Figure 1.1 and 1.2) this will probably be a powerful tool to help development of future optimizations. The usefulness of this tool is probably more in the area of larger methods which are hard to comprehend without higher levels of structure.

5.2 Conclusions

Decompilation in general is a difficult problem. In this project a working decompiler has been designed and implemented. The purpose was to visualize what optimizations actually do with the code inside the JVM, and for that perfect legal Java source code was not needed. With the extensions of goto statements and explicit locking and unlocking, the output of the decompiler is still fairly readable by humans. Some more improvements would be welcome though, and suggestions have been presented.
References


Appendix A

Syntax Tree Nodes

A.1 Tree nodes

The syntax tree representation is made up of the following node types.

- **block**: A basic block represents a group of nodes which constituted a basic block in the CFG. It has children constituting the contents of the block. Every kind of block node also keeps track of whether its label has to be visible.

- **exp**: An expression contains an expression tree without assignments.

- **ass**: An assignment expression is an expression assigning a value to a variable.

- **if**: An if node contains an expression tree as the condition and is also a block node containing children constituting the true path.

- **else**: An else node must be following an if node and is a block node containing children constituting the false path.

- **while**: A while node contains an expression tree as the loop condition and is also a block node containing children constituting the loop body.

- **goto**: A goto node contains a reference to its target basic block.

- **return**: A return statement without a return value.

- **retexp**: A return statement with a return value represented by an expression tree.

- **throw**: A throw statement contains an expression tree to represent the throwable object.

- **try**: A try block is a block node containing children constituting the contents of the try block.

- **catch**: A catch block may only occur after a try block or another catch block. It contains a class type specifying what kind of exception it handles and the exception variable. It is also a block node containing children constituting the contents of the catch block.

- **do**: A do node contains an expression tree as the loop condition and is also a block node containing children constituting the loop body.
switch A switch block contains a switch variable, case blocks and one default block.

case A case block contains a constant value to represent the case. It is a block node containing children constituting the body. It can only occur inside a switch block.

default A default block is a block node containing children constituting the body. It can only occur as the last child of a switch block.

break A break node is a break statement and has a reference to a parent block node which it breaks.

cont A continue node is a continue statement and has a reference to a parent loop which it continues.

sync A synchronized node is a synchronized block and contains an expression tree to represent the synchronized object. It is a block node containing children constituting the body.

lock A lock node represents explicit locking on an object. It contains an expression tree representing the synchronizing object.

unlock An unlock node represents the explicit unlocking of an object. It contains an expression tree representing the synchronizing object.

A.2 Expression Tree Nodes

Expressions make up their own tree structure. They consist of the following node types.

var A local variable.

type An object type.

bin A binary operator. It has two child expressions.

un A unary operator. It has one child expression.

field A field in an object. Contains either an object instance or a class depending on whether the field is static.

call A method call.

init A constructor call.

list A list of expressions. Used in calls for parameter lists.

cast A type cast. Casts an object instance from one type to another.

scast A simple cast. Casts a variable from one simple type to another.

scall A simple call. Used for some special method calls which are not real method calls in JRockit. (For example Math.abs(int))

array An array variable.
alloca A constructor call for array allocation.

sfield A simple field. Used for some special fields which are not fields in JRockit. (For example array.length)