Bachelorarbeit

# Optimizing Java Bytecode for Embedded Systems

ausgeführt am
Institut für Technische Informatik
der Technischen Universität Wien

unter der Anleitung von
Univ. Ass. Dipl. Ing. Dr. techn. Martin Schöberl

durch

Stefan Hepp
Kierlingerstrae 72
3400 Klosterneuburg

Klosterneuburg, am 24. Februar 2009

.....

**Abstract**

Modern Java Virtual Machines (JVM) for desktop and server computers use just-in-time (JIT) compilation to increase their performance. For embedded Java processors, JIT usually is not feasable. Therefore the java bytecode needs to be optimized for a specific platform ahead-of-time.

To generate optimized bytecode for the JOP Java processor several existing tools were compared. In order to implement optimizations for embedded Java processors a framework called JOPtimizer was created. Using this framework a method inlining optimization was implemented which honors the restrictions of the target platform.

# Contents

# 1 Introduction

The Java Virtual Machine (JVM) specification [3] defines an abstract architecture which was developed to safely execute compiled programs on different real hardware platforms. The binary format of the instruction code for the JVM is called *bytecode* and stored in `class` files. Each `class` file represents a single class and contains informations about its name, superclass, fields and methods, the bytecode for the methods and so on. Because the class files are used to distribute programs over the internet to a remote client, the format of the class files and of the bytecode is designed to be very compact and to allow a secure execution of the program.

Compilers like the `javac` compiler from Sun or from IBM create Java `class` files from source code written in the Java programming language, but compilers for other languages like ADA (GNAT [7]) or Python (Jython[1]) exist.

A bytecode program can be run by emulating the JVM and interpreting the bytecode instructions. To improve the execution time (after a certain warmup phase) just-in-time compilation in combination with profiling based optimization is usually used. As an alternative the platform independent bytecode can be compiled to a real hardware platform using an ahead-of-time compiler like GCJ[2].

Usually the java bytecode compilers only perform very few optimizations themselves and leave more advanced optimizations as well as platform specific optimizations to the JVM JIT compiler. Although JIT compilers have higher constraints on the execution times of the optimizations they can achieve better optimizations as ahead-of-time compilers due to profiling data aquired from the current execution of the program [9]. The bytecode can also contain all informations needed for debugging because all optimizations which might remove debugging informations can be run after the bytecode has been created.

As an alternative the JVM can be implemented as a real hardware processor. For this work the Java processor JOP [6] was used. JOP is a small processor for low cost FPGAs which implements the JVM instruction set and can be used in embedded systems. To run a Java program on JOP, Java `class` files are created using a compiler like `javac`, which are then packaged into a single `jop` file containing all the informations and the bytecode needed to run the program using the JOPizer tool [5]. The `jop` file can then be uploaded to the JOP processor where it will be executed.

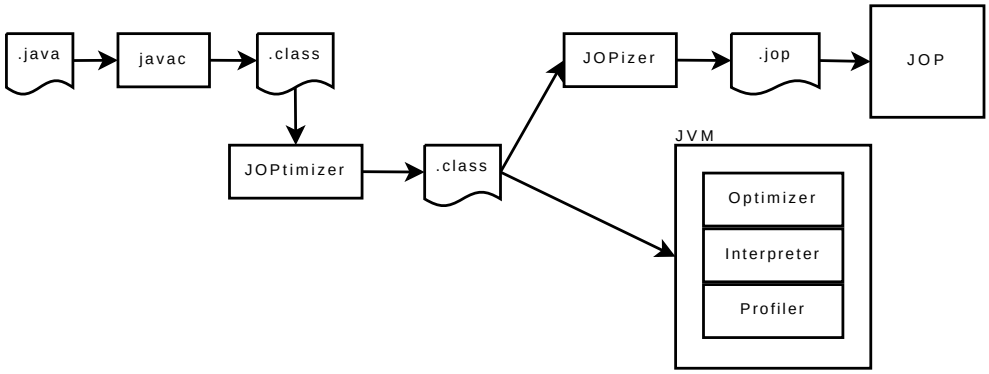
Using JIT optimizations in an embedded processor like JOP is not feasable. Therefore, ahead-of-time optimization has to be used in order to get a better execution time. In Section 2 some existing tools are presented. A new

[1] `http://www.jython.org/`
[2] `http://gcc.gnu.org/java/`

bytecode optimizer tool called JOPtimizer was implemented because the existing tools were not designed for embedded processors and the results were not satisfying.

JOPtimizer reads Java `class` files as created with compilers like `javac` and stores the optimized code in `class` files again, which makes it simple to integrate it into an existing toolchain. Figure 1 shows the toolchain used to execute a Java program on JOP or on a desktop JVM using JOPtimizer.

Figure 1: JOP Toolchain with JOPtimizer




The JOPtimizer framework and its design decisions are described in Section 3. The implemented optimizations, most importantly inlining, are presented in Section 4. Some benchmark results and a comparison with existing tools are presented in Section 5.

# 2 Existing Solutions

At first, several existing opensource bytecode optimization tools and frameworks were evaluated. To compare the tools, the benchmark application described in section 5.1 was compiled using Sun's `javac` and the results were used as inputs for the optimizers.

**ProGuard** ProGuard[3] is primarily intended as code obfuscator and code shrinker. It removes unused code and files, and is able to obfuscate code by renaming classes, fields or method names. It processed the benchmark bytecode within seconds and was able to reduce the code size by more than a one fourth, but no significant speed improvements were gained. When a more recent version of ProGuard was used, bytecode optimizations had to be disabled to prevent ProGruard from generating methods which exceed the maximum method size.

**Soot Java Optimization Framework** Soot[4] [12, 11] is a research framework designed for Java optimization which provides different intermediate representations and implementations of typical analyses and optimizations. It also features a whole-program mode which enables more aggressive intraprocedural transformations and optimizations. The soot framework took about one minute to optimize the benchmark application. The results of the optimized benchmark were improved only in one case, all other test programs were not improved or even showed a decreased performance. Additionaly enabling the whole-program optimization mode increased the performance by of all application benchmarks by about 11% compared to the previous results due to the invocation overhead of the JOP core.

**BLOAT** BLOAT[5] [4] is an academic Java bytecode optimization tool. It implements algorithms such as constant and copy propagation, loop peeling and loop inversion using a SSA form intermediate representation of the bytecode. It was one of the most promising of the evaluated programs, but with 165 seconds to optimize the benchmark application also the slowest. The generated class files could not be opened by the BCEL library used in the JOP toolchain, therefore no comparable results are available for BLOAT because the optimized benchmark could not be downloaded to the JOP processor.

[3] `http://proguard.sourceforge.net/`
[4] `http://www.sable.mcgill.ca/soot/`
[5] `http://www.cs.purdue.edu/homes/hosking/bloat/`

**GCJ** The gcc compiler suite includes an ahead-of-time compiler for the Java language and for bytecode class files, which is able to produce native code but also Java class files. However, when gcj is used to create bytecode class files, it does not read already compiled Java class files, therefore the benchmark was compiled and optimized using the Java source files. When the benchmark and the JVM sources of the JOP repository where compiled using only gcj, the resulting benchmark application crashed immediatly after startup on the JOP core. Therefore the JOP JVM classes were compiled using `javac` from Sun and only the benchmark application itself was compiled by gcj. The execution of this benchmark terminated after completing some tests because the gcj compiler emitted a *jsr* instruction which is not implemented in the JOP core. The completed tests showed a slight decrease in performance. The `-fwhole-program` optimization flag has no effect on the gcj compiler when bytecode is generated.

**JoGa, Jode** JoGa[6] and Jode[7] [2] are two tools with a graphical user interface which can be used to remove unused code and debugging informations, similar to ProGuard, but implement nearly no real code optimizations. Processing the complete benchmark application with these tools did not work, therefore no benchmark results are available.

The results of the benchmarks of ProGuard and the Soot framework are given in Table 3 in Section 5.

Most of the evaluated opensource tools are merely code shrinker or code obfuscators. The results of the remaining frameworks did not prove satisfactory and the frameworks provide no simple means to adept the optimizations for the limitations of an embedded systems Java core, like a maximum method size or a maximum stack size. Other compiler frameworks are usually not implemented for stack architectures. Therefore a new framework was created with focus on embedded Java cores, which is described in Section 3.

[6] `http://www.nq4.de/joga/`
[7] `http://jode.sourceforge.net/`

# 3 Framework

To implement the optimizations a framework called JOPtimizer was created. It handles the configuration and execution of optimizations and analyses, loads and writes class files and provides class hierarchy informations, intermediate representations of the code of the methods and classes to create a callgraph of the loaded application.

## 3.1 Assumptions and Design Decisions

In order to focus the implementation efforts some assumptions and design decisions were made. The optimizer was built primarily for embedded Java processors, which has a number of consequences.

- Only the Java Virtual Machine (JVM) architecture [3] is supported. The JVM instruction set is quite different to most other instruction sets. Implementing an intermediate representation based on the Java architecture, restricts the framework to Java, but simplifies many analyses and optimizations.

  The optimizer works on bytecode only which makes it independent from the source code. Therefore the application can be written in different languages (like Java, ADA, Eiffel, ...) as long as there exists a compiler for this language which is able to generate Java class files. It is also possible to run the optimizer on an application or library without having access to the original source files.

- The target architecture is known at design time. The code is not shared between different architectures. This allows the compiler to optimize the code for a specific microarchitecture with respect to parameters like the cache size or the execution time of bytecode instructions on the given target processor. The generated code is still in bytecode, but executing the optimized program on a different target microarchitecture may result in a lower speedup.

- All class files available at compile time (except native classes) and no dynamic class loading is used. If the complete application code can be loaded at once and no other classes will be added, replaced or removed after the compilation, then more virtual method invocations can be resolved and inlined. This is usually true for embedded systems and especially for the JOP processor, as the complete application must be downloaded to the processor as one single file. All software implementations of classes which are implemented as native classes on the target machine are ignored by the optimizations.

- No reflections, no Class.forName().newInstance() are used and all classes accessed by native code are known. This allows the calculation of the transitive hull of the application by following all class references in the code. Unused methods, functions, constants and classes can be removed to save memory. Like dynamic class loading, these features are usually not used in embedded systems applications.

Further design decisision and requirements of the framework include:

- The generated code will not be used to debug the application. Therefore debug informations like variable names or line numbers are not kept by the optimizations, as a correct mapping of the binary code to the source code is not always possible. For example, an inlined method code may need to refer to lines in another source file, which is not possible with the standard `LineNumberTable` attribute in Java class files.
- It should be possible to use the framework for worst case execution time (WCET) analysis and optimization. Due to this requirement the stack form of the intermediate representation described in section 3.3.2 was designed so that every instruction of the intermediate representation can be mapped to a bytecode instruction in an unambiguous way, so that the exact size and execution time of every instruction in the intermediate representation can be calculated. A basic support to preserve custom attributes of basic blocks was implemented.
- Different hardware configurations and different Java cores should be supported. The basic parameters of the target core like RAM access cycles or the maximum supported stack size are stored in configuration files. Plugins are used to calculate parameters like the instruction cycles or the method cache miss penalty for a given code size which are then used by the inlining optimization to estimate the speedup of the optimization.
- The hardware imposes additional restrictions. Depending on the target machine, the optimizations must take some limitations into account, like the maximum method code size, limited by the JOP method cache or the maximum number of variables and the maximum stack depth of a method, limited by the JOP hardware, which are considerably lower than the restrictions specified for the JVM.
- Exception handling is assumed to be rare in WCET oriented embedded systems. Exception handlers are still generated by the compiler for synchronized code to ensure that the monitor of a section is unlocked even when a critical section is left due to an exception. Hence exception handler need to be supported, but generating optimized code which includes

exception handlers was given a low priority. At the current state, generating exception tables in JOPtimizer is not fully implemented. Methods which have an exception table are therefore currently not optimized.

- Java subroutines are not supported. The JVM Specification [3] defines subroutines within methods, primarily used to implement `finally` exception handler. Current versions of the `javac` compiler from Sun do not emit `jsr` instructions since the verification of code containing such instructions is quite complex. The Java SE 6 verifier does not allow the use of these instructions[8].

  If a subroutine call is found, it could be inlined by replacing the call with a copy of the subroutine. However at the moment, like with exception tables methods containing subroutines are left untouched by JOPtimizer.

Some of the above assumptions can be disabled by command line flags. If not all referenced classes are found in the class path, missing classes can be ignored, which implies that dynamic class loading is used. If dynamic class loading is forced to be assumed, methods are not inlined if it is possible to overload them by loading additional classes which extend the existing classes.

If a reference to the reflection package is found, unused code should not be removed. Additional root classes which are not found in references in the class files can be specified manually so that the correct transitive hull of an application can be found.

Packages can be marked as library code, which tells the optimizer that classes in the library can be extended by classes in the application, but the library classes will not extend application classes. Thus even if classes of the library are missing during optimization, virtual methods can be safely inlined when the application code is known completely.

## 3.2 Class Loading

For each class, the Java compiler creates a class file which contains all informations about this class. This includes the classname, the superclass name, the names of the implemented interfaces, the fields of the class with names and signatures and the methods of the class with names and signatures. Classes, fields and methods have flags which define the visibility of the element, and, amongst others, if the element is static or final. Methods have additional attributes, which hold the bytecode as byte array, the exception tables, debug informations and custom informations. The Java class file format is defined by the JVM specification [3].

[8] `https://jdk.dev.java.net/verifier.html`

Each class also has a constant pool, which stores all class references as fully qualified class name strings. It also contains all strings, signature strings, names and all other constant values which need more than two bytes. All instructions which use strings, signatures, references or large constants store indizes into the constant pool. During execution an implementation specific runtime constant pool is created which is derived from the constant pool but keeps the resolved references instead of the symbolic references.

To find all classes which are used by an application, the transitive hull of the classes of the application is created. This is done by loading all classes which are referenced in the constant pools of the loaded classes and are not already loaded. This is repeated until no new classes are found. Then all references of all loaded classes are loaded. The root classes for this algorithm are the class which contains the `main` method as well as all classes which are referenced by native code. The `main` method must be specified as parameter of the optimizer, the names of the other classes can be set in a configuration file. If reflection is used, additional root classes can be specified.

Like other tools in the JOP toolchain, JOPtimizer uses the BCEL[9] library to read and write Java class files.

## 3.3 Code Representation and Transformations

The JVM has a stack architecture with variable-length instructions [3]. The format of the code of a method in a Java class file is called bytecode. An overview of Java bytecode is given in Section 3.3.1.

To overcome some drawbacks with the bytecode representation of the code, it is transformed to a new set of instructions which is called stackcode in JOPtimizer. This representation has fewer different but fully typed instructions but still operates on a stack. Each bytecode instruction in the original code translates to a single stackcode instruction and vice versa (with the exception of four pseudo-instructions used to mark local variables) while all information about the original bytecode instructions is preserved.

As described in [12], the stack architecture makes optimizations more complex, because the algorithms must take the effects on the stack and the implicit data dependencies of statements when instructions are moved, inserted or deleted into account. Therefore, like Soot, JOPtimizer provides a quadruple form code representation, which is called quadcode in JOPtimizer.

The quadcode form has even fewer different instructions and the stack is eliminated. Standard analysis and optimization techniques can be used without having to worry about a stack, but the number of instructions and

[9] `http://jakarta.apache.org/bcel/`

thus the bytecode size and the exact execution time depend on the number of load and store instructions which need to be inserted to create stackcode ouf of quaduple code again. If required it would be possible to further transform the quadruple code into static single assignment (SSA) form. This, however, is not yet implemented in JOPtimizer.

Figure 2: Transitions between different code representations

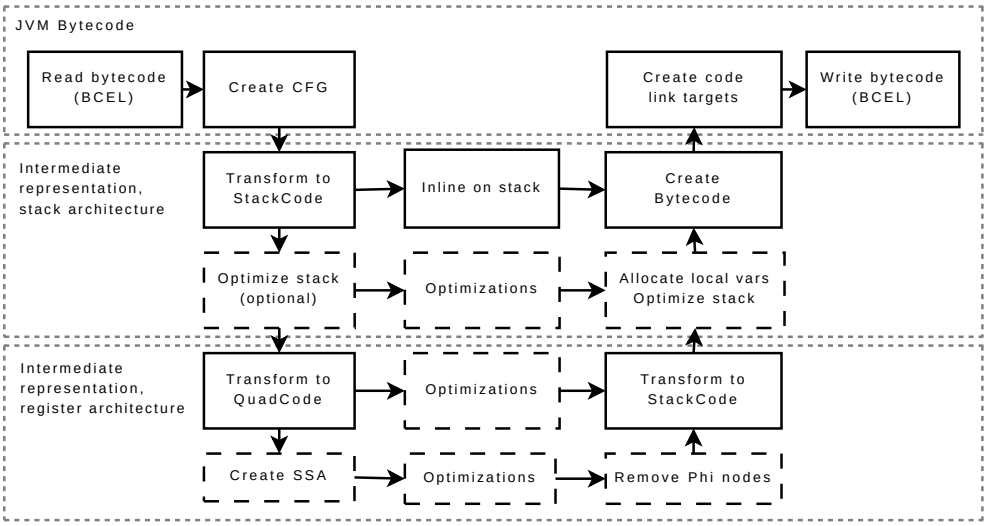



Figure 2 gives an overview of the different code form transformations and optimizations. Boxes with dashed lines represent functionality which has not yet been implemented in JOPtimizer.

The Soot framework implements all those different code representations, transformations and optimizations, but as it would be difficult to reuse only parts of the framework and the generated code was not optimal for an embedded processor like JOP, those features were reimplemented in JOPtimizer. However with some exceptions most of the basic principles and design features of the different Soot code representations as presented in [12] were used.

### 3.3.1 Bytecode

Java bytecode has about 200 instructions to load values from local variables or constants to the stack and store the top of the stack back to local variables and to perform simple arithmetic and logic operations, but also to load values from class fields, create new objects or to invoke methods. The basic types of values are integer, long, float, double and references. Double and long types are 64 bit wide, all other types are 32 bit wide. Indirect jumps, converting a

Listing 3.1: An example method and its bytecode

```
public int calc(int a, int b) {
    aload_0
    getfield 'Test.fField'
    fconst_2
    fdiv
    fstore_3                // float t1 = this.fField / 2.0f;
    iload_1
    iload_2
    fload_3
    f2i
    imul
    iadd
    istore 4                // int t2 = a + b * ((int)t1);
    iload 4
    ireturn                 // return t2;
}
```

reference to an integer or vice versa is not allowed which makes it simple to create a control flow graph (CFG) for bytecode.

Most instructions are typed and the instruction type must match the type of the operands on the stack, else the code will be rejected by the bytecode verifier. The variables and the entries on the stack are not typed and are always 32-bit wide. It is possible to store a 64-bit wide to a slot where a 32-bit value has been stored previously, but then the consecutive 32-bit slot will be occupied by the 64-bit value too. The compiler has to make sure that this slot will not be used while the 64-bit value in the previous slot is life. If a 64-bit value is loaded onto the stack, two slots are used. This must be considered when untyped instruction like `swap` or `dup` are used which operate on 32-bit slots only. Java types which are smaller than 32-bit like `short` or `byte` are always stored in 32-bit slots. Listing 3.1 shows the bytecode of a simple Java method.

Other features of the Java instruction set include exception handling, synchronization using monitor objects, and subroutines within methods. Constant values which are up to two bytes long can be stored with the instructions to load the constant. Larger constant values like strings or floats are stored the constant pool. Some instruction have a wide variant to store an index into the local variable table, the constant pool or a bytecode offset larger than 8-bit.

Exception handler ranges can start and end at any instruction in the code. To make the implementation of exception handling easier, like in Soot in the JOPtimizer framework basic blocks in the CFG are splitted at the beginning

and the end of exception handler ranges too so that each basic block is either completely covered by a range or not covered at all.

The stack depth at the end of all predecessor blocks of a basic block must be equal. The predecessors of a block define the stack depth prior to the execution of the first instruction in that block. It is therefore not possible to have different stack depths for different execution traces which therefore allows for a static stack analysis.

Reading and writing of bytecode is handled by BCEL in JOPtimizer. BCEL does some simple bytecode transformations on its own, like choosing between a normal and a wide variant of an instruction depending on the size of the used index or the constant, which must be taken into account when the exact opcode of an instruction is needed to determine the execution time or the number of bytes of that instruction.

The bytecode is parsed on demand. When the code of a method is modified and an optimization for a method is finished, the new code is compiled and stored as byte array again.

### 3.3.2 Stackcode

Instead of bytecode, a slightly different stack form is used in JOPtimizer, similar to the Baf code in Soot. Each bytecode instruction is mapped to a single stackcode instruction. Different variants of bytecode instructions are mapped to the same stackcode instruction with a type parameter and if applicable an variable index, a constant value or a class-, field- or method-reference in order to simplify the instruction set. A complete list of all stackcode instructions is given in Table 1.

In contrast to bytecode all instructions are typed and no differentiation is made between 32-bit values and 64-bit values. To determine the type of the parameters of stackcode instructions for untyped bytecode instructions like `swap` or `dup` a stack emulation is performed by traversing the CFG with a depth-first-search (DFS) and simulating the effects on the stack of the visited instructions. All references into the constant pool are replaced by their corresponding constant value, indices into the local variable table are used as index into a stackcode variable table.

In the JVM it is not valid to access the two consecutive 32-bit slots of a 64-bit variable separately. Therefore it is always possible to identify each 32-bit value as well as each 64-bit value with a single variable in stackcode. For instance a bytecode `pop2` instruction could be translated to a stackcode `pop.T[]` instruction with either two 32-bit types or one 64-bit type as parameter, depending on the content of the stack. The stackcode of the example method from Listing 3.1 is shown in Listing 3.2.

Table 1: List of stackcode instructions

| | |
|---|---|
| push @retAddress<br>push @exception<br><var> = @this<br><var> = @param | goto <target><br>ifcmp.<cmpop> <target><br>ifzero.<cmpop> <target><br>lookupswitch.<matches> |
| arraylength<br>arrayload.T[]<br>arraystore.T[]<br>newarray.T<br>newmultiarray.T.<dimension> | return.T<br>tableswitch.<low,high><br>throw |
| binop.T.<op><br>convert.T1.T2<br>iinc <var>,<inc><br>neg.T | load.T <var><br>push.T <const><br>store.T <var> |
| checkcast <class><br>getfield <field><br>instanceof <class><br>invoke.<itype> <method><br>new <class><br>putfield <field> | dup.T1[].T2[]<br>pop.T[]<br>swap.T1.T2<br>jsr <target><br>ret <var><br>monitorenter<br>monitorexit<br>breakpoint<br>nop |
| <itype> → interface,special,static,virtual<br><op> → +,-,*,/,%,<<,>>,>>>,^,\|,&,cmp,cmpl,cmpg<br><cmpop> → ==,!=,<,<=,>=,> | |

Listing 3.2: The example method as stackcode

```
public int calc(int a, int b) {
    l0 = @this
    l1 = @param.0
    l2 = @param.1
    load.ref l0                 // aload_0
    getfield 'Test.fField       // getfield 'Test.fField'
    push.float 2.0f             // fconst_2
    binop.float.div             // fdiv
    store.float l3              // fstore_3
    load.int l1                 // iload_1
    load.int l2                 // iload_2
    load.float l3               // fload_3
    convert.float.int           // f2i
    binop.int.mul               // imul
    binop.int.add               // idiv
    store.int l4                // istore 4
    load.int l4                 // iload 4
    return.int                  // ireturn
}
```

The local variable indices in the bytecode instructions are used as indices for the stackcode variables. However, if a 64-bit value is stored in a stackcode variable, it is not necessary to reserve the consecutive variable too. As those indices are used as indices into the local variable table of the bytecode when the stackcode is transformed back, either all used code transformations must preserve a valid stackcode variable allocation or a variable allocation step has to be performed prior to generating the bytecode to create a valid variable mapping where the consecutive slot is not used when a variable holds a 64-bit value.

At the entry point of methods, exception handlers and subroutines additional pseudo-instructions are inserted to make the assignment of parameters or the loading of the reference to the current exception or the return address of a subroutine explicit. These instructions are only used to set the initial state of variables and are removed when stackcode is transformed back to bytecode.

As already noted, each stackcode instruction can be mapped directly to a bytecode instruction making it possible to determine the execution time and the size in bytes of the code which will be generated but the generated instructions depend on the size of the constant values and constant table indices, the variable table indices and the target instruction indices.

### 3.3.3 Quadcode

The stackcode can be transformed to a quadruple form code if needed to eliminate dependencies between instructions due to the stack model. In Soot the quadruple form is called Jimple. A simple method to transform stackcode into quadruple code is to assign each stack entry a new variable and then to transform each stack instruction into one or more quadcode instructions using the variables assigned to the current top of the stack as parameters and return value. This creates a lof of unnecessary copy instructions which have to be removed by optimizations.

The quadcode has even less different instructions as the stackcode, because all instructions needed solely to manipulate the stack like `push` or `load` can be replaced by copy instructions. `pop` instructions can be removed. Table 2 contains a list of all quadcode instructions. Listing 3.3 shows the code of the previous example method transformed to quadruple form.

Although all instructions are typed, the variables are not. To determine the type of all variables, a type analysis has to be performed. As an alternative, the code could be transformed to SSA form [1] which allows for a simpler type analysis.

A straightforward method to transform the quadcode back to stackcode is to load all parameters of each instruction onto the stack and to store the result

Table 2: List of quadcode instructions

| | |
|---|---|
| <var> = @retAddress<br><var> = @exception<br><var> = @this<br><var> = @param | binop.T.<op> <var>,<v1>,<v2><br>convert.T1.T2 <var>,<val><br>neg.T <var>,<val> |
| arraylength <var>,<arr><br>arrayload.T[] <var>,<arr>[<i>]<br>arraystore.T[] <arr>[<i>],<v><br>newarray.T <var>,<size><br>newmultiarray.T.<dimension><br><var>,<size>[] | goto <target><br>ifcmp.<cmpop> <val1>,<val2><br><target><br>ifzero.<cmpop> <val> <target><br>lookupswitch.<matches> <val><br>return.T [<val>]<br>tableswitch.<low,high> <val><br>throw <exception> |
| checkcast.<class> <ref><br>getfield.<field> <var>,<ref><br>instanceof.<class> <var>,<ref><br>invoke.<itype>.<method><br>[<var>],[<ref>],<params>[]<br>new.<class> <var><br>putfield.<field> <ref>,<var> | copy.T <var>,<val><br>jsr <target><br>ret <var><br>monitor.<enter\|exit> <var><br>breakpoint<br>nop |
| <itype> → interface,special,static,virtual<br><op> → +,-,*,/,%,<<,>>,>>>,^,\|,&,cmp,cmpl,cmpg<br><cmpop> → ==,!=,<,<=,>=,> | |

Listing 3.3: The example method as quadcode

```
public int calc(int a, int b) {
    l0 = @this
    l1 = @param.0
    l2 = @param.1
    copy.ref s0, l0                 // load.ref l0
    getfield.'Test.fField' s0, s0   // getfield 'Test.fField'
    copy.float s1, 2.0f             // push.float 2.0f
    binop.float.div s0, s0, s1      // binop.float.div
    copy.float l3, s0               // store.float l3
    copy.int s0, l1                 // load.int l1
    copy.int s1, l2                 // load.int l2
    copy.float s2, l3               // load.float l3
    convert.float.int s2, s2        // convert.float.int
    binop.int.mul s1, s1, s2        // binop.int.mul
    binop.int.add s0, s0, s1        // binop.int.add
    copy.int l4, s0                 // store.int l4
    copy.int s0, l4                 // load.int l4
    return.int s0                   // ireturn
}
```

of the operation back from the stack to the result variable. Copy elimination on the stack code is needed to remove unnecessary `load` and `store` operations.

Another method used by Soot to transform quadruple code to stack code is to merge instructions back to full statements, basically recreating a high level source code, and then compiling the code into stackcode or bytecode, which creates lot less `load` and `store` instructions. Tests using the JBE benchmark on JOP as described in Section 5 showed that this method produces slightly better results than the first method, but in both cases the generated code was not optimal for JOP because Soot did not always generate the faster short bytecode instruction variants when it would have been possible.

# 4 Optimizations

## 4.1 Inlining

The most important optimization for the JOP core was method inlining in order to reduce the invocation overhead. In order to improve the execution time of the complete application, the speedup gained by eliminating the invocation overhead must be higher than performance decrease due to a higher cache miss penalty of the invoker, as the codesize of the invoker and therefore the time needed to load the invoker into the cache is increased.

Before a method can be inlined, a number of checks have to be performed to see if inlining is feasible and beneficial. Virtual invokations must be resolved first. Inlining itself can be done fairly simple by replacing the invocation with a copy of the control flow graph of the invoked method, adjusting the used variables and replacing the return instructions with jumps.

Inlining uses the stack intermediate representation. Although the implementation of the inlining algorithm can handle both representations provided by the framework, the stackcode provides a more accurate estimation of the resulting codesize, the number of used variables and the maximum stack depth because the quadruple form does not contain any information about the stack and the method code size depends on the number of `load` and `store` instructions which need to be inserted when the code is transformed back to stack form.

### 4.1.1 Callgraph Traversal

The speedup achieved by inlining depends on the order by which the methods are inlined. Inlining a method eliminates the invocation overhead but on JOP the method cache miss penalty for the invoker is increased because the code size of the invoker is increased.

A common heuristics which is also used by JOPtimizer is to traverse the method call graph in a bottom-up order. Therefore leaf methods which are usually executed more frequently are inlined first. The topological sort of the call graph can be established using a depth-first search [8]. Traversing the sorted method list in reverse leads to top-down inlining. In the benchmark setup presented in Section 5.1 bottom-up inlining always achieved better results than top-down inlining on JOP, for both the execution speed as well as the program code size.

Another approach may be to determine all invocations which can be inlined for the whole application using the checks presented in Section 4.1.2 and 4.1.3 before any inlining takes place and then sorting all invocations using the weight

function presented in Section 4.1.4. When a method is inlined the list of invocations which can be inlined as well as the weight of those invocations has to be updated for all edges in the call graph incident to the method which has been modified. However this heuristic is not yet implemented in JOPtimizer and therefore has not yet been tested.

### 4.1.2 Method Invocation Resolving

For each invocation the invoked method must be determined unambiguously. In JOPtimizer this devirtualization is done by reading the method reference stored with the invocation instruction and then by checking if any method overwrites the referenced method or if more than one possible implementation exists if the referenced method is an abstract method by searching the subclasses in the class hierarchy.

If dynamic class loading is used or not all classes of the application are known, only final, static or private methods can be inlined as other methods can be overloaded when a new class is loaded. If only classes marked as library classes are not analyzed, non-final methods can be devirtualized nevertheless as long as no library class extends a non-library class.

To get better devirtualization results type inference could be implemented to deduce the class of the invoked method more precisely.

### 4.1.3 Checks

When the invoked method has been determined, a number of checks have to be performed to find out if inlining is possible. Most of the checks which need to be done are presented in [11].

At first some preliminary checks are performed:

1. Abstract and native methods cannot be inlined. Synchronized methods are currently not supported by JOPtimizer.
2. Recursive invocations are not inlined.
3. A class or a package name can be excluded from inlining using the configuration. The invoker class as well as the class of the invoked method is checked.
4. The invoker must be able to access the invoked method, i.e. the invoked method must be visible to the invoker. This should always hold in a valid Java program but is checked for completeness. If an invocation would violate this rule, the JVM would throw an exception, which would not be thrown if the method would be inlined.

5. A method will only be inlined if it is static, private or final, or the class is marked final, or if dynamic class loading will not be used by the application and all application classes are known to the optimizer.

JOPtimizer needs to perform some non-standard checks regarding the restrictions of the target platform which is usually not done by standard optimizers:

1. The sum of the size of the code of the invoker, the invoked method and the additional code which will be inserted by the inliner must be smaller than the maximum method code size determined by the target platform.
2. The number of local variables needed after inlining must be less than than the maximum number of local variables available on the target platform.
3. An additional maximum size for the method to be inlined can be defined by configuration to prevent large methods from being inlined.

Afterwards, the stackcode of the invoked method has to be inspected:

1. If the invoked method uses exception handler (which includes all methods with synchronized code as exception handler are used to ensure that the monitor object is always unlocked) it is not inlined as this is currently not supported by JOPtimizer. If an exception in the invoked method is thrown the stack of the invoked method is cleared. If the method would be inlined, the stack of the invoker would be cleared as well. If the stack prior to the invocation would contain values other than the parameters of the method, it would be necessary to store those other values in local variables and to restore the stack after the invocation.
2. For all referenced methods or fields by instructions of the *invoked* method the following checks have to be performed:
   a) The class of the referenced method or field must be available in order to perform the rest of these checks.
   b) The referenced method or field must be visible to the invoked method, i.e. the original code must be valid.
   c) The referenced method or field must be visible to the invoker into which the code will be inlined.
   d) If the previous requirement does not hold, the referenced field or method can be changed to be public. For fields this is always safe because fields cannot be overloaded.

In order to change the visibility of methods, again the complete class hierarchy must be known and dynamic class loading must not be used or the method must be marked as final. If any subclass of the referenced class contains a method with the same signature and either the referenced method or the method in the subclass is private, the visibility cannot be changed and inlining is not possible. If any subclass contains a package visible or protected method with the same signature and the referenced method is not private (i.e. it is protected or package visible), those methods in the subclasses have to be changed to be public too if the referenced method is made public.

Changing the visibility of methods or fields can be disabled in the configuration. Inlining of methods containing such references is then not possible.

3. In addition to that for all methods invoked with `invokespecial` by the inlined method the following checks have to be performed:

   a) If the method is not private (i.e. an invocation of a super method or of a constructor) it is not inlined because this is currently not supported by JOPtimizer.

   b) If the referenced method is private but needs to be and can be made public, the invocation type must be changed to `invokevirtual` in the inlined code.

All invocations which match all those criteria are stored in an inline candidate list.

### 4.1.4 Inlining Decision

After all invocation candidates have been determined, JOPtimizer selects the invocations which are most likely to improve the overall speed. In order to calculate the minimal number of cycles saved by inlining, it is assumed that each invocation and the return from the invocation is a method cache hit and the invocation of the invoker itself is a cache miss. The number of cyclesi $r$ needed to read one byte, the cycles per invocation with a cache hit $I_{hit}$ or a cache miss $I_{miss}$ respectively and the cycles for a return with a cache hit $R_{hit}$ are determined depending on the target platform. For JOP, the instruction timings are described in [5].

Furthermore the bytecode size $S_{inv}$ of the invoked method and the size $S_{\Delta}$ of the bytecode inserted in addition to the inlined method (nullpointer checks, parameter passing and so on) minus the size of the removed invocation instruction are calculated using the stackcode instructions. The number of

expected invocations of the method to be inlined with a cache hit $c_{hit}$ and a cache miss $c_{miss}$ per execution of the invoker method are currently estimated using a simple heuristic. A better code analysis, profiling information or information gained by a WCET analysis will be needed to improve these estimations.

Inlining a method removes the cycles needed for the invocation but increases the cycles needed to load the invoker in case of a cache miss due to the increase of the code size. The expected gain for inlining a method which is invoked with $c_{hit}$ cache hits and $c_{miss}$ cache misses and $p_{miss}$ cache misses per invocation of the invoker is therefore

$$Gain = I_{miss} * c_{miss} + I_{hit} * c_{hit} + R_{hit} * (c_{miss} + c_{hit}) - r * (S_{inv} + S_{\Delta}) * p_{miss} \quad (1)$$

To lower the increase of program code, the final weight $W$ is calculated depending on the number of call sites of the inlined method $n_{callsites}$ by

$$W = \frac{Gain}{{n_{callsites}}^{0.2}} \quad (2)$$

The exponent is an empiric value.

The method with the highest weight is selected for inlining. Invocations with a negative weight are removed from the candidate list. After the method has been inlined all new invocations in the inlined code are added to the list of inlining candidates and their weight is calculated. This process is repeated until no inlining candidates are left.

### 4.1.5 Method Inlining

To inline a method, the basic block containing the invocation is split at the invoke instruction and the instruction is removed. A null pointer check is inserted if the invoked method is not static. The control flow graph of the invoked method is copied and inserted into the CFG of the invoker. All return instructions in the control flow graph are replaced by jumps to the new basic block which contains the instructions after the originial invocation instruction. If the method returns a value, it has to be placed on the stack by the preceding instructions. Therefore when the new `goto` instruction is executed, the return value is on the top of the stack, which matches the semantics of the invoke instructions.

To ensure that the stack size after the inlined method is correct, the stack size previous to each return instruction in the invoked method must be calculated. This can be done by traversing the CFG with a DFS and calculating a new stack size after each instruction from the previous stack size. If the stack before a return contains values other than the return value, the values below the top of the stack must be removed by temporarily storing the top value

Listing 4.1: Method with two values on the stack before return

```
public static int test(int i, Object o) {
1:  iload_0
2:  aload_1
3:  ifnonnull 6      // if ( o == null ) {
4:  iconst_0
5:  ireturn          //     return 0;
                     // }
6:  ireturn          // return i;
}
```

in a local variable and inserting pop instructions. Rearranging the previous code or eliminating instructions which create unused values so that the pop instructions can be omitted can be left to other optimizations.

An example of a method which has other values than the return value on the stack is shown in Listing 4.1. Sun's `javac` compiler does not emit such code because after each statement in the original code all values on the stack are stored in the local variables which correspond to the variables in the original source code.

The stack values of the invoked method are simply placed above the stack of the invoker. The stack values of the invoker are not changed, because the stack depth of the invoked method must never be negative. The sum of the maximum stack depth of the invoked method and the stack depth at the time of the invocation minus the size of the parameter values must not exceed the maximum stack depth of the target machine, else the method cannot be inlined, at least not without inserting spill code to save the stack of the invoker into local variables.

The local variables of the invoked method are mapped into the local variable array of the invoker. A simple mapping is to add the maximum number of used local variables of the invoker as offset to the indices of the local variables of the invoked method. All instructions which access local variables like `load` and `store` need to be updated in the inlined code. This ensures that the invoked method will not destroy the content of any active variable of the invoker.

All methods which are not inlined recursively can use the same offset for the local variables because they are only used within the CFG subgraph of the inlined method. As the subgraphs of two or more inlined method do not overlap, the live ranges of the variables do not overlap either, so the same local variables of the invoker can be used for different inlined methods. If methods are inlined recursively, a safe offset for the local variables of the inner inlined method is the sum of the offset used for the surrounding method

and the maximum number of local variables of the original invoker of the inner method. The sum of the offset and the maximum number of local variables of the inlined method must not exceed the maximum number of local variables of the target machine, else the method cannot be inlined, or a better mapping of the local variables has to be found by searching for local variables of the invoker which are not live at the time of the invocation.

The parameters are passed to the inlined method by removing them from the stack and storing them into the local variables which are mapped to the first local variables of the inlined method with `store` instructions.

## 4.2 Peephole Optimizations

Additionally, three simple peephole optimizations were implemented. The first optimization removes all `nop` instructions. If the instruction is a target of a branch or a jump, the target is redirected to the next instruction. Another optimization removes all unneccessary `goto` instructions to the next instruction.

The last peephole optimization replaces a sequence of a `putstatic` followed by a `getstatic` instruction which store and read a value of a static class field by a `dup` and a `putstatic` instruction if the same field is accessed and if it is not volatile, thus replacing a `getstatic` instruction with a faster one.

# 5 Benchmarks

## 5.1 Benchmark Setup

To compare the different existing frameworks and to evaluate the speedup gained by the implemented optimizations, the Embedded Java Benchmark (JBE) from the JOP sources was used.

It consists of several small applications which are repeatedly executed. The execution time for executions is measured by the benchmark application. The code size informations of the JOP download tool were used to compare the impact of the optimizations on the codesize. The tool provides the sum of all method bytecode instruction words as well as the total RAM needed for the downloaded application, which includes the bytecode size as well as the size of the constant pools and exception tables and other class informations.

The benchmark consists of three sets of tests. The first set measures the execution times of several single bytecode instructions. This is done by comparing the execution times of two functions, which only differ by the one instruction which is to be measured. Although the execution time depends entirely on the Java core, the soot framework produces quite different code even if only a single instruction is added and therefore distorts the results because the execution times of two completely different methods are compared. Because this test is designed to measure only the performance of the hardware, its results were ignored.

The second set tests the performance of the JOP core. It contains the Sieve benchmark which tests the performance of a single method which uses some control flow instructions and array instructions.

The third group contains three benchmarks Kfl, UdpIp and Lift, which are three small Java applications with several methods and classes. These benchmarks offered the greatest potential for optimizations within this test.

To optimize the code, the ProGuard optimizer and the Soot framework introduced in Section 2 as well as the own JOPtimizer tool where used.

Soot was evaluated with various options. First with the standard optimizations enabled (*-O*), and a second time in whole-program optimization mode (*-W*). Both modes were evaluated again, but with the *-via-shimple* option enabled, which causes Soot to generate code using the Shimple intermediate representation instead of creating and compiling expressions. The generated code is slightly larger and has a slightly lower performance in all tests, therefore the results using the Shimple-option are not included in the final comparison.

The benchmark application was compiled to class files using `javac` of the Sun JDK 1.6. The generated files were then processed by the various

Table 3: Results of JBE Benchmark

| Benchmark | ProGuard | Soot (-O) | Soot (-w -W) | JOPtimizer |
|---|---|---|---|---|
| Bytecode size | **-26.2%** | -2.4% | 6.3% | 12.4% |
| External RAM | **-26.1%** | -1.2% | 3.5% | 6.9% |
| Execution Time | 3s | 62s | 61s | 5.5s |
| Sieve | **1.1%** | -2.7% | -2.8% | 0.0% |
| Kfl | 0.1% | 0.0% | 11.2% | **13.3%** |
| Udplp | 2.1% | 3.8% | **15.7%** | 8.9% |
| Lift | 1.0% | -5.3% | 7.8% | **14.0%** |

optimization tools and results were uploaded to the JOP core to measure the results. The JOP core was synthetisized using the default settings and was run at 60Mhz.

## 5.2 Benchmark Results

The results of the benchmarks are summarized in Table 3. The biggest speedup was achieved by using extensive inlining. This is hardly surprising because an invocation on the JOP core needs over 70 cycles. A method cache miss results in additional cycles to load the method code into the cache.

The costs of the inlining optimizations are a bigger codesize and a higher cache miss penalty. Therefore the gain of the method inlining depends on the resulting method size and the cache miss rate.

In some tests the optimized code generated by Soot even decreased the performance. This is partly due to a bad instruction selection because the instruction cycles of the different variants on the JOP core of the instructions was not taken into account. The performance decrease was compensated by the speedup achieved by inlining in whole-program-optimization mode.

JOPtimizer works best on small methods which are not overloaded and therefore can be devirtualized and inlined, which is the case with most getter and setter methods. Inlining larger methods is limited by the maximum method code size and may have a negative performance impact, depending on the cache misses of the invoker. A better estimation of the number of executions of the invocations is needed to improve the results, when larger methods are inlined.

To counteract the increase of the codesize due to inlining, inlined methods which are not invoked anymore should be removed. When ProGuard was used to shrink the output of JOPtimizer (which is about 12% larger than the reference), the optimized benchmark bytecode size was reduced by 26% which in total results in a 14% smaller codesize compared to the reference

Table 4: Results of an application benchmark (execution time in seconds)

| Benchmark | Reference | Optimized | Speedup |
|---|---|---|---|
| Interpreter | 15.61 | 13.17 | +18% |
| JIT | 6.98 | 6.95 | +0.4% |

benchmark. It is possible and in view of these results desirable to include an optimization step to remove unreferenced classes, methods and fields as well as dead code elimination in a future version of JOPtimizer.

## 5.3 Application Benchmark

To test the optimizer with a larger desktop application, JOPtimizer has been used to create an optimized version of JOPtimizer itself. Only the JOPtimizer application itself has been optimized, libraries (BCEL, log4j) and library code from the JRE were excluded. The application consists of 272 classes. 105 methods were inlined by JOPtimizer.

The original program as created by the Sun Java compiler `javac` version 1.6 and the optimized JOPtimizer were executed with the same input (the unoptimized JOPtimizer application) on the Sun JVM 1.6, the first time using only interpretation of the bytecode (`java -Xint`), the second time using the default mode with just-in-time (JIT) compilation enabled.

The mean execution times needed to load, analyze and optimize the code are given in Table 4. The time needed to write the class files to the disk is not included because this step only uses methods from BCEL and depends strongly on the IO speed. Writing the class files took about 10 seconds both in interpreter mode and in JIT mode.

When the bytecode was interpreted only, the optimizations performed by JOPtimizer were able to increase the execution time by about 18%. With JIT compilation enabled, as expected no significant speedup was achieved.

# 6 Conclusion

Existing opensource bytecode optimizers do not take the additional restrictions imposed by the limited resources of embedded processors and the real execution times of instructions into account, which can lead to a degredation of performance or to the creation of methods which are too large or use too many variables to be executed on the target platform.

JOPtimizer implements an inlining optimization which is aware of the target platform limitations. A speedup of up to 14% has been achieved in the benchmarks. Inlining larger methods must be done carefully because the invocation time on a method cache miss increases and may even result in a slower execution. The total bytecode size of the application also increases significantly due to inlining, therefore in a future version of JOPtimizer an optimization to remove unused code should be included.

To implement other optimizations a local variable allocator and copy elimination for the stack code are needed. More sophisticated optimizations can then be implemented using a quadruple form or SSA form of the code and therefore do not need to take the effects on the stack when instructions are deleted or inserted into account. Realtime embedded systems will need optimizations which target the worst-case execution time instead of the average execution time.

# Bibliography

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[2] Jochen Hoenicke. Jode: Java optimizer and decompiler.

[3] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification*. Prentice Hall PTR, second edition, April 1999.

[4] Nathaniel John Nystrom. Bytecode-level analysis and optimization of java classes. Master's thesis, Purdue University, 1998.

[5] Martin Schoeberl. *JOP Reference Handbook*. Number ISBN 978-1438239699.

[6] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Number ISBN 978-3-8364-8086-4. VDM Verlag Dr. Müller, July 2008.

[7] Edmond Schonberg and Bernard Banner. The gnat project: a gnu-ada 9x compiler. In *TRI-Ada '94: Proceedings of the conference on TRI-Ada '94*, pages 48–57, New York, NY, USA, 1994. ACM.

[8] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java, Part 5: Graph Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[9] Mark Stoodley, Kennath Ma, and Marius Lut. Real-time java, part 2: Comparing compilation techniques. April 2007.

[10] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.

[11] Vijay Sundaresan. Practical techniques for virtual call resolution in java. Master's thesis, McGill University, Montreal, 1999.

[12] Raja Valle-Rai. Soot: A java bytecode optimization framework. Master's thesis, McGill University, Montreal, 2000.

[13] Bill Venners. *Inside The Java Virtual Machine*. McGraw-Hill Companies, second edition, January 2000.