FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Worst-Case Execution Time Driven Method Inlining for Embedded Java Processors

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Stefan Hepp
Matrikelnummer 0026640

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Privatdoz. Dipl.-Ing. Dr.techn. Martin Schöberl

Wien, 01.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Stefan Hepp
Kierlingerstraße 72
3400 Klosterneuburg

"Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Stefan Hepp Wien, am 15. November 2011

Abstract

Java bytecode is commonly executed by a Java virtual machine (JVM) on a desktop computer, which usually either interprets the code or employs a just-in-time (JIT) compiler to translate the code to native machine code. Profile-guided optimizations are used to speedup hotspots at runtime. In contrast to that, Java processors like the Java Optimized Processor (JOP) execute Java bytecode directly without the need for a JIT compiler or an interpreter.

For real-time applications the worst-case execution time (WCET) is more important than the average-case execution time (ACET). If the WCET of a real time task is too high, ahead-of-time optimization is required that focuses the worst-case execution path to reduce the WCET.

In this thesis, we show how the worst-case analysis (WCA) can be used to guide the optimizer and discuss the impact of a method cache on the performance of the optimizations. An existing WCET analysis tool is used to drive WCET oriented optimizations for the JOP processor. The main optimization is inlining method calls to eliminate the invoke overhead, while trying to avoid increasing the overall WCET due to higher method cache miss penalties. A framework has been implemented which allows the optimizer and the WCA to work on the same data structures in memory so that the WCA can be invoked repeatedly during the optimizations with low overhead.

To test the impact of the optimizations, various benchmark applications have been optimized and analyzed, and the WCET, the ACET as well as the code size of the optimized applications are compared to their unoptimized version.

Zusammenfassung

Java Bytecode wird üblicherweise in einer Java Virtual Machine (JVM) auf einem Desktop PC ausgeführt. Meistens wird der Code von der JVM interpretiert oder mittels eines Just-In-Time (JIT) Compilers in Maschinencode übersetzt. Basierend auf Profiling-Informationen werden kritische Stellen zur Laufzeit bei Bedarf optimiert. Java Prozessoren wie der Java Optimized Processor (JOP) können hingegen Java Bytecode direkt ausführen ohne zusätzlich einen Interpreter oder JIT Compiler zu benötigen.

Die Worst-Case Execution Time (WCET) ist bei Echtzeitsystemen wichtiger als die Average-Case Execution Time (ACET). Falls die WCET eines Echtzeit-Prozesses zu hoch ist, werden Optimierungen während der Entwicklung benötigt um den Worst-Case Pfad im Programm zu optimieren und damit die WCET zu reduzieren.

In dieser Diplomarbeit wird gezeigt, wie eine Worst-Case Analyse (WCA) verwendet werden kann um einen Optimizer zu unterstützen. Des weiteren werden die Auswirkungen des Method Cache auf die Effizienz von Optimierungen betrachtet. Ein bestehendes WCET Analysetool wird verwendet um Optimierungen für JOP zu lenken. Als zentrale Optimierung wird Method Inlining verwendet, wobei versucht wird eine Verschlechterung der WCET durch höhere Cache-Miss Kosten zu vermeiden. Ein Framework wurde erstellt um die Optimierungen und die Worst-Case Analyse auf den selben Daten arbeiten zu lassen, damit die WCA während der Optimierung mit geringen Overhead mehrfach durchgeführt werden kann.

Um die Auswirkungen der Optimierungen zu testen wurden mehrere Benchmark-Anwendungen optimiert und analysiert. Die WCET, die ACET und die Codegröße der optimierten Anwendungen wurden mit den nicht optimierten Versionen verglichen.

To my mother,
who is able to remember the title.

Acknowledgements

First of all I would like to thank my supervisors Priv. doz. Dr. Martin Schöberl and Univ.Prof. Dr. Peter Puschner for their valuable feedback and their patience and support that made this thesis possible.

I would also like to thank Wolfgang Puffitsch and especially Benedikt Huber for all the lengthy discussions that spawned many ideas for further improvement, as well as for providing the analysis tools that have been used in this thesis, and for letting me hack their code to integrate it with the optimizer.

Last, but not least, I owe my deepest gratitude to my family for their support that made it possible for me to work on this thesis, and to my friends who helped me to keep my sanity all along the way.

Contents

Chapter 1

Introduction

An embedded system needs to interact with its physical environment. Usually, the system needs to meet deadlines imposed by the environment when it reacts to external events, or it needs to execute a task periodically at predefined intervals. Such systems are called real-time systems. An important property of those systems is the worst-case execution time (WCET). It is the maximum execution time of a given program or part of a program for any possible input data. The WCET of the tasks of a real-time system must be low enough so that they always meet their deadlines.

A worst-case analysis (WCA) can be used to calculate a bound on the WCET of a given part of an application. A precise calculation of the WCET is not possible in most cases due to the large number of program states to explore and the complexity of the behavior of the underlying hardware. The WCA requires bounds on the execution time of all operations, and requires bounds on the number of iterations of all loops and recursions. Hints like loop bounds that restrict the feasible paths are called flow facts. In the general case, manual annotation is required to provide such loop bounds, but a flow analysis can be used to derive some of the required flow facts automatically.

If the WCET is too high, i.e., if the application may not always meet the required deadlines, the system designer either needs to use a faster processor, which is usually more expensive and has a higher power consumption, or he needs to optimize the application. A compiler may perform optimizations to reduce the execution time of the application. In contrast to conventional applications however, the optimizations need to reduce the worst-case execution time, not the average-case execution time (ACET).

Java programs are usually executed on a Java virtual machine (JVM), which is executed on a register-based architecture.. The JVM either interprets Java bytecode instructions or compiles the Java code to the native architecture before it executes the application. Desktop JVMs usually use profiling information collected at runtime to optimize hot spots.

Java processors on the other hand implement the Java virtual machine in

hardware. The Java Optimized Processor (JOP) [22, 23] is a small Java processor designed to simplify the WCET analysis. Its native instruction set called microcode is designed for efficient execution of Java bytecode. The processor executes bytecode instructions by invoking a sequence of microcode instructions for every bytecode instruction. The timing of the bytecode instructions is composable, i.e., the execution time of a sequence of instructions is always equal to the sum of the execution times of the individual instructions. JOP uses a novel instruction cache called method cache that caches whole methods. Instruction cache misses can only appear at invoke and return instructions, all other instructions are guaranteed cache hits.

Due to the limited resources of small embedded processors like JOP, advanced optimization at runtime based on profiling data are difficult to achieve and can introduce additional runtime overhead. Furthermore, optimizing the application using profiling information collected at runtime only improves the execution speed after several iterations of the code and thus does not decrease the WCET, but increases the complexity of the WCET analysis.

For real time systems we therefore need ahead-of-time (AOT) optimizations to reduce the WCET. The target architecture needs to be known in detail at design-time, as this is required by the WCET analysis to calculate a good WCET approximation. The optimizer can also use this information to optimize the code for the concrete target architecture. Feedback from a WCET analysis can be used to guide the optimizations. The WCET path is independent of specific executions and can therefore be calculated statically and used by an ahead-of-time optimizer. Runtime profiling information is not needed to improve the results.

For this thesis a new framework for analyzing and optimizing Java bytecode was created, and an existing data-flow analysis and an existing WCET analysis tool have been ported to the new framework. A WCET-driven optimizer called JCopter has been implemented on top of this framework, which performs method inlining using feedback from the WCET analysis and a method cache analysis.

In Section 1.1 we will present the motivation for this thesis. Section 1.2 discusses related work. The rest of the thesis is structured as follows: Chapter 2 gives an introduction to WCET analysis, the JOP architecture, and some data structures used by the optimizer. In Chapter 3 we discuss various optimizations with respect to JOP, the method cache and the worst case execution time. Chapter 4 presents the implemented inliner and introduces a WCET driven algorithm to select call sites for inlining. We show how the optimizer interacts with the cache analysis and the WCET analysis. Chapter 5 describes the implemented framework and the toolchain and discusses some implementation specific issues. The results achieved by the optimizations are presented in Chapter 6. The thesis concludes with a summary and an outlook in Chapter 7.

1.1 Motivation

A compiler can perform optimizations to reduce the execution time or the code size of a program automatically. The developer does not need to optimize the code by hand, which would make the source code less readable. Instead the developer can focus on writing more structured code and on expressing the intended meaning in the source code (e.g., by using a multiplication instead of a shift operation to multiply with 2^n), and let the compiler eliminate redundancies and find semantically equivalent but faster versions of the code.

Furthermore the compiler can evaluate its optimization decisions anew every time the developer changes the code. This would be a significant burden to do manually. Performing optimizations by hand is also an error-prune process and requires additional code documentation to convey the intention behind an optimization to other developers. Also, the developer can only perform optimizations at source code level, while the compiler can optimize the binary code for the specific target platform.

In this thesis we focus on real time applications in which the WCET is more important than the ACET. Since optimizations targeted at the average case can have a negative effect on the WCET, we need WCET-driven optimizations instead. To avoid implementing a separate timing analysis within the optimizer, we use the existing WCET analysis to guide the optimizer. All execution timings and WCET related analysis results are provided by the worst-case analysis.

The main target architecture used for this thesis is JOP. As JOP uses a method cache, the cache costs in terms of time spent for filling the cache are directly related to the code size of methods. Cache miss costs can be a significant part of the execution time, therefore we need to consider the impact of the optimizations on the method cache. A method cache analysis is required to estimate cache costs.

Method inlining has been chosen as primary optimization for two reasons: First, the method invocation overhead on JOP is quite high, so removing that overhead can yield a significant gain. Secondly, when method inlining is performed automatically, the developer can use standard code patterns like getter and setter methods or wrapper methods to improve the modularity of the code with low or even no execution time penalties.

We have chosen to implement our own framework for two reasons: First, it allows us to port all existing tools to the same code base with minimal effort. This reduces the redundancy of code in the toolchain and allows for an efficient interaction between tools. Secondly, the WCET analysis requires the application code in its final bytecode form to perform a precise analysis. The Java optimization framework Soot [33] uses a quadruple code representation for code optimizations. Java frontends for the LLVM[1] [9] framework

[1]The Low Level Virtual Machine Compiler Infrastructure Project, `http://llvm.org/`

and the GCC[2] compiler exist, but the internal low level code representations are too different from the Java architecture. Translating the internal code representation back to Java bytecode after optimization is infeasible. Instead we use any standard `javac` compiler to generate Java bytecode, and perform the analyses and optimizations entirely on Java bytecode. A transformation between different code representations is not required.

1.2 Related Work

In the last decades, a significant amount of research has been done regarding automatic code optimizations. Most of the developed techniques target the ACET and the code size of the applications to optimize. Only few works exist that explicitly target the WCET of an application. As the gap between the computational speed and the memory speed increases, it also becomes increasingly important to take the cache costs caused by optimizations into account. The method cache used by JOP is a comparatively new cache architecture. To our knowledge, no previous work exists that investigates optimizations for target platforms that use a method cache.

Inlining is a well known optimization to reduce the calling overhead. The inliner developed in this thesis is similar to the algorithm described by Zhou et al. [37]. They present a fast inlining algorithm that tries to maximize the gain achieved by inlining, while keeping the application code size below a predefined limit. Selecting call sites to inline is based on a heuristic *rebate_ratio* that is calculated for all nodes in the call graph. The *rebate_ratio* is defined as the function calling frequency divided by the expected code size increase when that function is inlined. However, the algorithm does not take instruction cache costs into account, and it is not designed to target the WCET.

The TU Dortmund WCC optimizer[3] [12, 18] is a C compiler for the Infineon TriCore TC1796 and TC1797 processors that implements several WCET analysis driven optimizations. The aiT WCET analyzer is used for timing analysis and to find the worst-case execution path (WCEP). WCC first translates ANSI-C source code into a high-level intermediate representation, that is used by most code optimizations. A code selector then generates a low-level intermediate representation (LLIR) from the high-level intermediate representation. Additional optimizations are performed on the LLIR.

The aiT WCET analysis[4] interfaces with the WCC framework on the level of the LLIR code. The results of the WCET analysis are mapped back to both the LLIR as well as the high-level code representation using back-annotation, so that they can be used by the optimizations. Evolutionary algorithms have been used to find good optimization sequences that improve either both

[2]The GNU Compiler Collection, `http://gcc.gnu.org/`

[3]`http://ls12-www.cs.tu-dortmund.de/research/activities/wcc/index.html`

[4]`http://www.absint.com/ait/`

WCET and ACET or both WCET and code size [12]. The authors found that most standard optimizations affect both ACET and WCET in a similar way for the TriCore architecture, which uses both a scratchpad instruction cache and a set associative instruction cache.

WCC also features a heuristic WCET-driven function inliner [11]. Call site selection is done by random forests. Random forests consist of several decision trees, a majority vote on a random subset of the decision trees decides whether a call site should be inlined or not. The decisions are based on various features such as method sizes or WCET analysis results. The decision trees are trained by supervised machine learning. The WCET analysis is used to calculate the WCET of several benchmark applications. For several call sites it is tested if inlining that call site improves the WCET. Together with the feature vectors of the call sites, those results are passed to the machine learner that generates the decision trees. This approach has the advantage that the inliner can be easily retrained for new target architectures. The compiler designer does not need to manually tune the heuristics used in the optimizer. The machine learning based inliner can outperform manually crafted heuristics.

Zhao et al. use interaction with a WCET analysis to apply path optimizations on the WCEP [36]. The WCEP-driven optimizations are applied after performing some traditional code optimizations. Superblock formation is used to copy the WCEP into a superblock, i.e., a sequence of basic blocks that has only one entry but can have multiple exits. Path duplication duplicates the WCEP within loops, while loop unrolling is used to duplicate whole loop bodies, regardless of the WCEP. The main advantage of those optimizations are that they may create additional optimization opportunities and that some transfer of control penalties within the loop bodies can be eliminated, due to rearranging the WCEP into sequential code. However, these optimizations increase the code size. The optimizations have been applied only to the innermost loops to limit the code size increase. Instruction cache costs have not been taken into account because the used target architectures does not use caches.

Java bytecode is usually optimized during execution on a Java virtual machine. The Java HotSpot VM is the high-performance desktop JVM reference implementation [16, 8]. In contrast to a traditional just-in-time (JIT) compiler that compiles every method when it is first executed, the HotSpot VM starts by interpreting the code first. When a hot spot is detected, i.e., when the JVM finds a piece of code that is executed very often and is therefore responsible for a large percentage of the execution time, the hot spot is compiled and optimized, using profiling information gathered during the initial executions of the code. Inlining is used to reduce the method invocation overhead and to create larger code blocks for the optimizer. Deoptimization is used when the class hierarchy changes due to dynamic class loading. Optimizations are performed on a high-level intermediate representation that is in static single-

assignment (SSA) form. However, the WCET of an application running on such a platform is hard to analyze.

Since Java programs are usually executed on JVMs that perform profiling-driven optimizations, most existing bytecode optimizers focus on unused code removal and on code obfuscation.

Soot is an ahead-of-time (AOT) Java optimization framework that uses a 3-address intermediate representation to facilitate code transformations [33]. It takes Java class files containing bytecode as input and transforms the code to an intermediate representation called Baf that is similar to bytecode. Then the code is transformed from stack code to a 3-address code (or quadruple form) intermediate representation called Jimple. The code can also be converted to SSA form, or to a more high-level intermediate form that represents the code as expressions. Optimizations are performed on the 3-address code [32]. To generate optimized bytecode, Jimple code can be translated back to Baf stack code that is then optimized. Alternatively, Jimple code can be aggregated into expressions, which can then be compiled efficiently into stack code without the need for additional optimizations. In any case, the resulting stack code is then compiled to bytecode using the Jasmin bytecode assembler. Soot performs several intra-procedural optimizations such as copy propagation, constant folding and dead code elimination, as well as method inlining. Inlining is performed in a bottom-up manner, the code size increase is limited by predefined code size bounds. However, the optimizations are not designed to target the WCET of the applications. Interaction with a WCET analysis requires transforming the code from 3-address form as used for optimization back to optimized stack code that is required for a precise timing analysis.

A ACET-driven Java method inliner for JOP has already been implemented [3]. In this thesis, the method inliner has been ported to a new framework, and it has been integrated with the WCET analysis tool and the data-flow analysis tool of the JOP toolchain. The feedback from the WCET analysis is used to implement the WCET-driven method inliner presented in this thesis.

Chapter 2

Background

This chapter discusses the background of this thesis. The notions of average-case execution time and worst-case execution time are discussed. A short overview over the Java virtual machine and the Java Optimized Processor is given, as those form the main target architecture used in this thesis. We also discuss the data-flow analysis and the worst-case analysis tool used in this thesis.

2.1 Worst-Case Execution Time

The worst-case execution path (WCEP) of a method is a control flow path with the highest execution time. The execution time of the WCEP is the worst-case execution time (WCET) of that code.

The task of a WCET analyzer is to find a safe and tight upper bound on the actual WCET of a method [35, 5]. To do this, the analysis must first find all feasible control flow paths. Hints that restrict the set of feasible paths are called flow facts. If the code contains loops or recursions, the number of iterations must be bounded. Flow facts that bound the number of loop iterations are called loop bounds. If any loop or recursion is unbounded, it is possible to construct control flow paths of infinite length. The execution time of such a path would be infinite.

Flow facts can be provided by the system designer in the form of code annotations, or they can be derived automatically from the source code by a flow analysis. For instance, if the loop variable of a `for` loop iterates from a constant initial value to a constant end value, the compiler can derive an upper loop bound directly from the source code.

A data-flow analysis can also be used to derive flow facts. It calculates an over-approximation of the possible values that the variables of a program can hold at specific program locations. This can be used to determine the value range of loop variables. Abstract interpretation can be used to implement a data-flow analysis. Abstract interpretation calculates abstract states for

every program location that over-approximate the set of concrete states that the program can have at that program location. The analysis starts with an empty abstract state for every location and then iteratively transfers an abstract state to a new abstract state, using the semantics of the instruction at the corresponding program location. The new abstract state is joined with the existing abstract state at the program location after the instruction. This is repeated until a fix-point is reached.

The precision of the abstract states determines the runtime of the analysis, therefore the abstract states should only capture the information the user is interested in, nothing more. On the other hand, if the abstract states are not precise enough, the analysis might not find any bounds on the values of the variables. If the data-flow analysis is unable to determine loop bounds, the programmer must provide loop bounds or he must use loop constructs that can be analyzed. Flow facts that are provided by source code annotations need to be transformed according to the code transformations performed during optimization and compilation, as the flow facts are required by the worst-case analysis that must analyze the generated code in order to get safe results.

If the flow facts are imprecise, the worst-case analysis might find a WCEP that is actually infeasible. This contributes to an over-estimation of the WCET.

The second task in a worst-case execution time analysis is to determine the actual execution time of the WCEP, i.e., the WCET of the analyzed code. The actual execution time depends on the properties of the target platform. However, the WCEP depends on the execution time of the code. Determining the WCEP first and then calculating the execution time of that path is therefore not possible, except if the code has only a single control flow path, i.e., when the control flow path is independent of the input data. Calculating the WCET by enumerating all paths and calculating the maximum execution time over all paths is usually not feasible due to the huge number of possible paths. A WCET analysis must therefore compose the WCET of a program from partial results of subgraphs of the control flow graph.

Modern processor architectures are very complex, which makes the estimation of the execution time of instructions very hard or even unsafe. Due to features such as pipelines, instruction and data caches, branch prediction and speculative execution and out-of-order execution, the execution time of an instruction or a sequence of instructions can depend on the processor state prior to the execution of the instructions. This brings about a potentially large over-estimation if the processor state at the beginning of the analyzed code segment is not known. Due to the large number of possible concrete processor states, the WCET analysis needs to use abstract processor states, which introduces non-determinism to the hardware model and renders an exact WCET analysis impossible.

Complex processors can also exhibit timing anomalies [13, 20]. A faster

execution of an instruction sequence can lead to a processor state that causes subsequently executed code to have a higher execution time than it would exhibit after a slower execution of the first instruction sequence. This increases the complexity of the composition of execution times of instruction sequences.

It should be noted that the WCEP depends on the target platform. The WCEP can be different for the same program for different target platforms. Therefore the results of the WCET analysis cannot be easily reused when a program is ported to a different platform. Also, the actual WCEP can be different from the WCEP found by the analysis due to the over-approximation of the execution timings, even if the path found by the analysis is a feasible path.

Two main methods exist to analyze the execution time of some code, measurement-based timing analysis and static timing analysis. A measurement-based timing analysis executes the code on the target hardware and measures the actual execution time, while a static WCET analysis calculates a bound on execution time of a code sequence for some initial abstract processor state based on a hardware model. The advantage of a measurement based analysis is that the analysis does not need a precise hardware model. However, to measure the execution time, some concrete input data must be used. One of the central problems of measurement-based analyses is that in the general case it is not known which input data and which initial processor states trigger the execution of the WCEP. On the other hand if the control flow is not data-dependent and if the part of the processor state that can influence the execution time of instructions is forced to a known state by special instructions at the beginning of the analyzed code, the result of the measurement is precise.

A static WCET analysis does not require access to the target hardware or concrete test input data, but it requires a very precise hardware model. Adapting a static WCET analysis tool for a new target architecture is therefore more time-consuming than for a measurement based analysis. The precision of the analyzed WCET bound depends on the precision of the hardware model and the complexity of the processor.

The WCET analysis used in this thesis is a static WCET analysis [28]. It uses abstract interpretation for data-flow analysis. The analyses are presented in more detail in Section 2.5.

2.2 The Java Virtual Machine

The Java virtual machine (JVM) is a stack architecture designed to execute object-oriented code written in the Java programming language. In this thesis, we restrict ourselves to the Java 6 JVM specification [10, 15].

Source code written in the Java programming language can be compiled to Java class files using the `javac` compiler provided by the Java development

kit. The class files can be executed by a JVM such as the Java HotSpot VM [16]. The HotSpot VM interprets the code and uses profiling information to detect, compile and optimize often executed pieces of the code at runtime.

At the time of writing, the Java 7 specification [17] has just been released, which introduces an additional `invokedynamic` instruction and additional constant pool entries to support dynamically typed programming languages in the JVM. Those features however are not suited for static analysis and are not supported by the JOP toolchain.

Class Hierarchy and Class Members

Java uses a single-root class hierarchy. The root object is `java.lang.Object`, all other classes are subclasses of that class. Multiple inheritance is not allowed, but classes can implement multiple interfaces. Interfaces themselves can also extend multiple interface. Classes are members of packages. A class can have fields and methods as class members. In Java subroutines are always associated with a class. Subroutines are therefore called methods, not functions as in non-object-oriented languages. Methods are always class members.

A class can also be nested inside an enclosing class. Classes are categorized in the following way:

- **Top-level class**: A class that is a direct member of a package, i.e., it has no enclosing class or enclosing method.
- **Nested class**: Every class that is not a top-level class is a nested class.
- **Static nested class**: A static nested class is always a member of the directly enclosing class. However it has no reference to an instance of the outer class.
- **Inner class**: A non-static nested class is called inner class. An inner class is created with a reference to an instance of the enclosing class. It has access to all members of that instance of the enclosing class. An inner class can be either a member of the directly enclosing class, or it can be defined within an enclosing method.
- **Local class**: A local class is an inner class that has a class name and is enclosed by a method.
- **Anonymous class**: An anonymous class is an inner class that has no class name and is enclosed by a method.

It should be noted that the class attribute that stores class nesting information is called `InnerClasses`, although it stores information about static nested classes too.

Classes and class members have an access modifier that defines which classes have access to it. Class members have access if the class of the member

has access to a class or class member. A class has access to all top-level classes that are either declared public or that are members of the same package.

A class member can have one of the following access modifiers:

- **Public**: Every class has access to that class member.
- **Protected**: Let C be the class of the class member. The class member can be accessed by C itself, any class in the same package as C, and any subclass of C.
- **Package visible**: Let C be the class of the class member. The class member can be accessed by C itself and any class in the same package as C.
- **Private**: Only the class containing the member has access to it.

The member name of a method (i.e., the name without a class name) together with its method descriptor (i.e, the representation of the types of the arguments and the return type) is called the method signature. A non-static, non-private method m overrides a method s if the class of m is a subclass of the class of s and if s can be accessed from m. An abstract method is a method without an implementation. A native method is a non-abstract method that is not implemented in Java. The implementation can be provided by an external binary library or by the JVM implementation.

Constructor methods are named `<init>` in the class file. Classes can also have a static initialization method named `<clinit>`, which is invoked exactly once when the class is accessed for the first time.

Non-static fields can be inherited, but they never override other fields.

The Java Class Files and Bytecode

The JVM specification defines the instruction set architecture and the file format of the binary class files. For every class in the application, a separate class file is generated, which contains information about the fields and methods of the class, references to the super class and the implemented interfaces, as well as a constant pool and the code of non-abstract, non-native methods defined in the class.

The constant pool contains all constants used in the class, such as strings and numbers. It also contains entries that reference classes, methods and fields. Those entries are used in the class to refer to the superclass and the implemented interfaces as well as by instructions that access fields and invoke methods. Entries are indexed by consecutive indices. However the index 0 is not a valid index, and the index after a `long` or `double` constant is defined to be unusable by the JVM specification.

The code in the JVM is called bytecode. Instructions take their operands from a stack and put the result back on the stack. Load and store instructions

```
void invoker() {          class A {
  A a = new B();              void m() { }
  a.m();                  }
}                         class B extends A { }
```

Figure 2.1: Declared type and receiver of an invocation

are used to load values from a local variable table on to the stack or vice versa. Every method on the call stack has its own local variable table. The local variable table of a method initially contains the arguments of the method. The maximum stack height and the maximum local variable table size are known statically. The JVM requires that the stack height for any instruction is always the same for every execution of the instruction.

The stack and the local variable table consist of 32 bit wide slots. Most bytecode instructions are typed. The types of values on the stack or in the local variable table can be 32 bit or 64 bit wide integer or floating-point numbers, or a 32 bit wide reference to an object or an array.[1] Bit-manipulations of references are not allowed by the JVM. Memory addresses cannot be stored, indirect jumps are not possible in the JVM.[2]

The bytecode instructions are of variable length. For some parametrized instructions a `wide` variant exists that adds additional bytes to the instruction to increase the range of the parameter. Among others, the JVM instruction set includes instructions to manipulate the values on the stack, to perform arithmetic operations on the stack and to load the value of fields on the stack or put the top of the stack into a field. The `new` instruction is used to create a new object of a given type and place a reference to the new object on the stack. Several instructions exist to invoke a method, as described in the next section.

Method Invocations

A method is invoked at a call site. We call the method containing the call site the invoker and the invoked method the callee.

The declared type of a call site is the class that is statically referenced by the call site. The receiver of an invocation is the object that receives the call, i.e., the object of which the call site invokes a method. The runtime type of a call site is the type of the receiver. In the example in Figure 2.1 the

[1] Besides the `int`, `long`, `float` and `double` types the JVM also defines `boolean`, `char` and `short` types. They are used primarily for arrays and are implicitly converted to `int` when they are put on the stack.

[2] The special `jsr` and `ret` instruction are defined by the JVM. They can be used to store the current process counter into the local variable table and to jump back to the stored location. This is primarily intended for the implementation of subroutines that handle exceptions, but the latest `javac` compiler does no longer emit those instructions.

declared type of the invocation `a.m()` is class `A`, while the receiver type of the invocation is class `B`. Class `B` inherits the invoked method `m()` from class `A`.

Java 6 defines four different invoke instructions. All invoke instructions have a reference to a method as parameter. The class of the referenced method is the declared type of the invocation. The descriptor of the referenced method defines the values that are consumed from the stack.

- `invokestatic`: This instruction is used to invoke static methods. In contrast to the other invoke instructions, this instruction does not consume an object reference, and the method referenced by the instruction is always the invoked method.
- `invokevirtual`: This instruction performs a virtual invocation by invoking the method that is defined in the receiver or inherited by the receiver. This can be a different method than the method statically referenced by the instruction. The declared type of the invocation must not be an interface.
- `invokeinterface`: The `invokeinterface` instruction is similar to the `invokevirtual` instruction except that the declared type of the call site must be an interface.
- `invokespecial`: This instruction is used to perform non-virtual invocations of non-static methods. This includes private methods, constructor methods and super methods, but it can also be used to perform a non-virtual invocation of any other method. Super method invocations are handled slightly different (see below).

To support recompilation of only a subset of the classes of an application or changing a library to a new version without recompiling the application, the `invokespecial` instruction is specified so that it may call a different method than the method it refers to. For instance if a class `C` extends a class `B` which in turn extends a class `A` and both `A` and `C` but not `B` define a method `m()`, a super call in `C.m()` would refer to `A.m()`. If class `B` would be recompiled at a later time so that it now has a method `m()` too, without recompiling class `C` the super call in `C.m()` would still refer to the super method in class `A`, but the Java specification requires the instruction to call `B.m()`.

2.3 The Java Optimized Processor

The Java Optimized Processor (JOP) is a small, single-issue stack processor for embedded systems [22, 23]. The processor implements a RISC architecture that is significantly simpler than the JVM. The native instruction set of the processor is called microcode. JOP executes Java bytecode by executing either a sequence of microcode instructions or by invoking a static Java method

for every bytecode instruction. The latter method is used to implement complex instructions like `new`, which needs to invoke a garbage collector, or to implement floating point operations in software.

To access hardware devices, hardware objects are used to map I/O registers to fields of singleton objects [26, 25]. The I/O registers can be accessed simply by using the fields of the hardware objects. Interrupt handlers can be registered to interrupt sources that can be triggered in hardware or in software [24]. The JVM implementation on JOP also provides several native methods to access memory directly. Invocations of such native methods are replaced by special bytecode instructions before the application is downloaded to the target. Those instructions are then executed by microcode like standard bytecode instructions.

A tool called JOPizer takes standard Java class files of the application and of a downscaled Java runtime library implementation and creates a `.jop` file that contains the complete code of the application in Java bytecode, as well as the part of the JVM that is implemented in Java, all constants and the class hierarchy information. The `.jop` file can then be downloaded to the processor, which starts by executing a startup method to perform initialization tasks such as executing the static class initialization methods `<clinit>` or setting up the scheduler and the garbage collector. After initialization, the entry method of the application is invoked by the startup method.

JOP has been designed from the beginning to simplify WCET analysis. The JVM uses `wait` instructions in the microcode to stall execution of a bytecode instruction until a read or write operation performed by the microcode of that instruction is completed. Processor resources are not shared between bytecode instructions.

Similarly, the processor does not perform speculative execution at branches. Instead, branch bytecode instructions simply clear the pipeline by executing an appropriate number of `nop` instructions in the microcode. Since the pipeline of JOP is very short, the branch costs are very low nevertheless.

As a result, there is no interlocking at bytecode level. The execution time of every bytecode instruction that is implemented in microcode can be calculated statically if the memory access delays are known. It is independent of the processor state and thus of the execution history (with the exception of instructions that access the cache, but the cache miss costs can be calculated statically too). The execution time of a sequence of bytecode instructions or of basic blocks can therefore be calculated simply by adding up the execution times of the individual instructions or basic blocks. There are no timing anomalies due to pipelining effects, which simplifies the WCET analysis.

The JOP architecture restricts the number of local variable slots and the maximum stack size to a much lower value than the JVM specification. However since the maximum number of used local variable slots and the maximum stack size are static properties, the tools can check at design time if the con-

straints are met.

The Method Cache

JOP employs a novel instruction cache called method cache [21] or function cache [7]. When a method is invoked, then the whole method is loaded into the cache if it is not already in the cache. Similarly, an invoker is completely loaded into the cache when the invoked method returns if it was removed from the method cache during the execution of the invoked method.

This cache organization has the advantage that instruction cache misses only appear at invoke and return instructions. All other instructions are guaranteed cache hits, which simplifies the WCET analysis. The cache content depends on the sequence of method invocations, the size of the invoked methods, and on the replacement policy, but not on the addresses of the cached instructions. This also simplifies code generation, because it makes code positioning considerations to avoid cache line conflicts obsolete.

Caching whole methods also means that the cache miss costs of a single invoke are always proportional to the size of the invoked method, not to the fraction of the code that is actually executed of that method. A large method that exits early, e.g., due to some parameter check, or that always executes only a fraction of its code, e.g., if it contains a large `switch` statement, can cause large cache miss costs for a comparatively small number of executed instructions. Similarly, if a method first executes the first half of its code sequentially, then invokes a method that causes it to get removed from the cache, and then executes the second half of its code after the return from the invoked method, the whole method has to be loaded into the cache twice. However, the instructions are executed only once, even if the method does not contain any loops. On the other hand, invoke and return instructions hide part of the cache miss costs on JOP. For sufficiently small methods the execution time on a cache miss and a cache hit of an invoke or return instruction can be the same. Therefore, to keep the total cache miss costs of an application down, the methods should be kept small.

The cache is organized in blocks of fixed size. The number of cache blocks influences the cache lookup costs. A method is mapped to a set of consecutive cache blocks. The size of a method must not be larger than the size of the cache.

For up to two cache blocks a Least-Recently Used (LRU) replacement policy can be used, i.e., on a cache miss the cache block with the oldest access time is replaced. For more than two cache blocks this becomes impractical to implement. If the oldest cache entry should be replaced by a method larger than the oldest cache entry, the second-oldest entry must be removed too. In this case the processor would either need to defragment the cache content so that the new method can be placed into a continuous cache region, or it would need to place the method into several non-continuous regions, which increases

the complexity of the instruction fetch.

Instead a First-In First-Out (FIFO) policy is used. This means that the oldest entry in the cache is replaced, even if it has been used more recently than other entries. The disadvantage of that policy is that it is more difficult to analyze and does not perform as well as the LRU replacement policy [28, 19].

In general, a cache access at an invoke or return instruction can be classified using the following categories [7]:

- **always miss**: Every cache access is a miss. Since cache miss costs are never lower than cache hit costs on JOP, this is always a safe upper bound on the cache costs.
- **at most one miss**: During the execution of a given scope of the program, there is at most one cache miss, every other access is a cache hit. The total number of cache misses for the execution of the whole program is bounded by the number of times the program can enter the scope for which the cache access is classified as *at most one miss*.
- **persistent**: The first cache access after the application entered a given scope is a cache miss, every following access is a cache hit until the application leaves the given scope. This is more strict than *at most one miss* which does not require the cache miss to appear before the cache hits. The upper bound on the number of cache misses for persistent cache accesses is the same as for *at most one miss*.

  If a cache access is persistent during the execution of the whole program, it is called globally persistent, else it is called locally persistent.
- **always hit**: Every access is a cache hit.
- **not classified**: The cache access cannot be classified as any of the above categories.

A data-flow analysis can be used to classify cache accesses [7]. The existing worst-case analysis uses the fact that if a method m accesses at most $maxBlocks(m) \leq N$ distinct cache blocks of a FIFO or LRU cache with N blocks during its execution, then every access of that execution can be classified as *at most one miss* [28]. The number of blocks required for all methods reachable from m in the call graph, including m, is a safe upper bound for $maxBlocks(m)$. If $maxBlocks(m) \leq N$, the worst-case analysis classifies all cache accesses within m as *at most one miss*, else as *always miss*.

If the cache uses an LRU replacement policy, all cache accesses at invoke instructions in methods with $maxBlocks(m) \leq N$ can be classified as *persistent* (assuming that the cache is not fragmented), return cache misses can be classified as *always hit*. For FIFO caches, this is not the case. Assume that a method m with $maxBlocks(m) \leq N$ invokes a method a in a loop, and that the cache contains m as its oldest entry and a as its second-oldest entry. The

first invoke of a is therefore a cache hit. If a invokes a method that is not yet in the cache, the FIFO cache will replace the oldest entry, in our case m, with the invoked method. When a returns, m is no longer in the cache and needs to be loaded, replacing the cache blocks holding method a. The second invoke of a is therefore a cache miss. All following cache accesses to a and m are cache hits as long as method m does not return, since they are now the newest entries in the cache and are replaced only when more than N distinct cache blocks are accessed.

The cache miss costs for a FIFO cache can be higher than for LRU caches, since the cache misses of methods reachable from a method m with $\mathit{maxBlocks}(m) \leq N$ can also appear at return instructions, which hide fewer cache load cycles than invoke instructions.

It is possible to force a FIFO cache to behave like an LRU cache in the analysis by inserting instructions into the code that explicitly clear the cache. If such instructions are inserted at all locations where the program enters a scope for which the analysis can show that at most N distinct cache blocks are accessed during execution, then every first access in those scopes is a cache miss since the cache contains no entries when it enters that scope. Therefore the analysis can classify cache accesses at invokes as *persistent* and cache accesses at return instructions as *always hit*, which simplifies the analysis and avoids the higher cache miss costs at return instructions. The actual number of cache misses can increase, since explicitly clearing the cache invalidates cache entries, which could otherwise still be used later on. This increases the ACET, but the more precise cache analysis can lead to a better WCET analysis result. However this has not been tested in the course of this thesis.

2.4 Call Graphs and Call Strings

A call graph is a directed graph that contains non-abstract methods as nodes. An edge between two nodes represents a call of the method at the head from a call site in the method at the tail of the edge. In the general case virtual invokes can have different receivers at runtime. This is represented by multiple outgoing edges to all possible implementations of the virtual invoke.

In Java, the receiver of a non-virtual invoke is always known statically. The set containing the declared type of a virtual invoke (i.e., the class that is statically referenced by the invoke instruction) and all subclasses of the declared type is a superset of all receivers of the invoke instructions. The set of method implementations inherited by those receivers of a call site contains all methods that can be executed by the invoke instruction. This is known as class hierarchy analysis [2].

The call graph can be constructed recursively by processing all new nodes. For every new node, outgoing edges to all method implementations of all call sites of the method are added. If no node for the invoked method implemen-

tation exists, it is added to the graph and added to the set of new nodes.

Call graph thinning techniques can be used to remove infeasible edges from the call graph. Rapid type analysis removes receivers that are never instantiated in the application, while variable type analysis tries to find all types that can reach a variable and can thus appear as receivers of call sites using the variable [31]. Alternatively, the results of the receiver type data-flow analysis can be used to find method implementations of call sites while constructing the call graph.

A path exists in the call graph from a method `a()` to a method `b()` if method `b()` may be executed during the execution of method `a()`. Therefore, if the program contains recursions, the call graph is cyclic. Vice versa, if the call graph is a directed acyclic graph (DAG), the application code is not recursive. On the other hand, it is not necessary that a recursion-free program has an acyclic call graph. This can happen if the application contains wrapper methods that invoke the same method of a different object that provides a different implementation. If the receiver analysis cannot show that the type of the receiver of the call site in the wrapper is always different from the wrapper itself, the call graph contains a self-loop at the wrapper method. Using the results of the receiver type data-flow analysis such self-loops can be usually removed though.

If the call graph is acyclic, a topological ordering can be derived. In a topological ordering every node i that has an edge to a node j appears before j in the ordering. Therefore, if the methods in a call graph are visited in topological order, then when a method m is visited, every method for which m may be invoked during its execution (i.e., every method that has a path to m in the call graph) has been visited before. If a call graph is traversed in reverse topological order, a method is only visited after all reachable methods have been visited. This property simplifies many algorithms that calculate analysis results based on the results of all invokers or of all invoked methods.

A depth-first search (DFS) algorithm can be used to find back edges in a call graph [29]. The set of back edges is not unique however, since the DFS can mark different edges as back edges depending on the order in which the direct successors of nodes are traversed.

Call strings are sequences of call site references and are used to make analysis results and the call graph context-sensitive. A call string $(c_1, c_2, \cdots, c_n)$ represents a sequence of method invocations at call site c_i. The method m_i containing the call site c_i is the method invoked at call site c_{i-1}. The call string corresponds to a path $(m_1, m_2, \cdots, m_n)$ in the call graph. If the call string starts with the entry method, it represents a single execution context of a method. Otherwise the call string contains the top entries in the call stack and represents all contexts that have that call string as its suffix.

A set of contexts of an execution of method m can be represented by the tuple $\langle c, m \rangle$, where c is a call string leading to the invocation of method m, i.e.,

the tuple $\langle c, m\rangle$ represents all contexts of method m where the top of the call stack equals c. In our implementation, the call graph is made context sensitive by using those tuples as nodes. If a call site c_m in a context represented by $\langle c, m\rangle$ invokes a method m', then the call graph contains an edge from $\langle c, m\rangle$ to $\langle c', m'\rangle$, where c' is equal to or a suffix of $(c_1, \cdots, c_n, c_m)$. If the length of the call strings is unbounded and increased by one at each call site, then the call graph is unfolded into a call tree. If the call graph is cyclic, the resulting call tree has an infinite number of nodes. By bounding the length of the call strings in the call graph by some constant k, the resulting graph remains finite since the number of call sites and therefore the number of unique call strings is finite. A context-sensitive analysis that only uses call strings of length up to k is called k-limiting context analysis. The context-insensitive call string can be represented as a special case of the call graph with call strings where all call strings are of zero-length [1]. In a context-sensitive call graph, each method can have a different set of successors per context.

The last entry c_n of a call string c of a node $\langle c, m\rangle$ represents the call site in method m_n that invoked the method m. To find all method implementations which can be invoked at a call site c_m in a context represented by $\langle c, m\rangle$, the following procedure can be used: Let $c'' = (c_1, \cdots, c_n, c_m)$. First, find all nodes in the call graph that represent a context of method m. For every directly reachable context $\langle c', m'\rangle$ check if one of c' and c'' is a suffix of the other one, and if so, add m' to the set of invoked methods.

If c' is a suffix of c'', then all contexts $\langle c'', m'\rangle$ are also represented by $\langle c', m'\rangle$, i.e., $\langle c', m'\rangle$ is less precise than $\langle c'', m'\rangle$. We therefore need to add m' although this might be an over-approximation. If c'' is a suffix of or equal to c', then the contexts represented by $\langle c', m'\rangle$ are a subset of the contexts for which we search the invoked methods, therefore m' needs to be added. By iterating over all nodes that can be directly reached from any context of m, we guarantee that we check all contexts where c_m appears as the last invoke.

It is also possible to use an edge-labeled call graph instead of creating separate nodes for different contexts. In this case the call strings are attached to the edges, not to the nodes. The structure of the graph only changes if a receiver of a call site is removed in every context of the invoker, the number of nodes is independent from the call string lengths. Finding the invoked method implementations of a call site in this graph can be done by checking the labels of the outgoing edges of the method containing the call site in a similar way.

It should be noted that a call site does not need to be an invoke instruction. Bytecode instructions that are implemented as Java methods in the JVM can also be represented in the call graph as a call site at the instruction invoking the JVM method. This can be used to find all unused methods not only in the application, but in the JVM as well. Since JVM methods are also loaded into the method cache, those methods are actually required to be represented in the call graph if the call graph is used by the cache analysis to determine

the set of methods that may be loaded into the cache during execution of a method.

2.5 The WCET Analysis Tool

The data-flow analysis (DFA) and the WCET analysis (WCA) presented in [28] are used in this thesis. They have been ported to the new framework (see Chapter 5) and interact over common data structures with the optimizer. The data-flow analysis provides a context-sensitive receiver-type and loop-bound analysis. The WCA employs a static WCET analysis to calculate the WCET of a given method. In addition to the loop bounds provided by the data-flow analysis, the WCA also reads loop bound annotations in the source code and merges them with the data-flow results.

The WCA expects a target method as parameter. The analysis then calculates an upper bound on the WCET of a single execution of the target method. The target method is usually an interrupt handler or a periodically executed real-time task. The analysis then constructs a separate call graph that has the target method as its root, which must be acyclic, i.e., there must be no recursive calls in the real-time code. All loops must be bounded, either by the data-flow analysis or by manual source annotations.

A microcode timing analysis is used to automatically derive the execution time for every bytecode instruction that is implemented in microcode on JOP. The timings of the bytecode instructions are composable and can be calculated once, since the execution time of bytecode instructions does not depend on the processor state by design (with the exception of invoke and return instructions, in those cases the execution time depends on the content of the method cache and the size of the invoked method).

The WCA can either use a WCET analysis based on the implicit path enumeration technique (IPET), or a model checking based WCET analysis [4, 28]. In this thesis we use the IPET-based analysis.

The IPET-based analysis transforms the control flow into an integer linear programming (ILP) problem. To do this, the analysis first calculates the execution time c_i of every basic block B_i using the bytecode instruction timing analysis. Virtual invokes are devirtualized by creating separate parallel paths at every virtual call site, where each path invokes a different possible callee.

The analysis then derives constraints for the execution frequencies e_i from the fact that the sum of the execution frequencies of the ingoing edges of a basic block B_i is equal to e_i. The same holds true for the outgoing edges of every basic block. The execution frequency of the entry block is set to 1, and the execution frequencies of back edges are bounded by the loop bounds.

The analysis can now use an ILP-solver to calculate the WCET of a control

flow graph with n basic blocks by solving

$$\text{WCET} = \max_{\langle e_1,\cdots,e_n\rangle \in \mathbb{N}_0^n} \sum_{i=1}^{n} c_i e_i$$

under the above constraints.

The recursive WCET analysis adds the WCET of methods to the execution time of their devirtualized call sites. Since the analysis requires an acyclic call graph, it can traverse the call graph in reverse topological order to calculate the WCET of all invoked methods before analyzing the invoker methods.

The global WCET analysis creates a supergraph by inserting the control flow graph of the invoked method into the call graph of the invoker after the call site of the invoked method, and solves the ILP-problem for all methods at once. This is a very precise analysis but the size of the problem to solve is much larger.

Cache costs are handled by creating two alternative paths for every devirtualized call site and for every return. One path is assigned the execution time of a cache hit, the execution time of the other path also includes the additional cache miss costs. The execution frequencies of the paths that represent the cache misses are bound by the cache analysis.

The WCET analysis provides several different cache approximations. In this thesis we use the following three:

- **always miss**: For every call site the number of cache misses is equal to the execution frequency of the call site, i.e., the execution frequencies of all cache hit paths are zero. This analysis provides a safe upper bound for both the cache costs and the analyzed WCET.

- **at most one miss**: If for a method m $\mathit{maxBlocks}(m) \leq N$ holds, where N is the number of method cache blocks, then the cache accesses in m are classified as *at most one miss*, otherwise they are classified as *always miss*. For *always miss*, the number of cache misses is bounded by the execution frequency of the basic block containing the cache access. For the recursive WCET analysis, cache accesses classified as *at most one miss* can have at most one cache miss per invocation of the method containing the cache access. For the global WCET analysis the number of cache misses is bounded by the number of times the program may enter the scope for which the cache access is classified as *at most one miss*, which usually results in a lower total number of cache misses.

- **always hit**: For all call sites the number of cache misses is zero, i.e., the cache costs are ignored by the analysis. The resulting WCET can be used to calculate the fraction of the WCET returned by an analysis using a different cache approximation that includes only cache miss costs.

The value of $maxBlocks(m)$ is determined by the set of methods reachable from m in the call graph.

Chapter 3

Optimizations

The main goal of this thesis is to implement optimizations that decrease the WCET respectively the computed WCET bound of programs executed on JOP or similar processors. Optimizing for JOP has two main advantages that are exploited in this thesis. First, we use bytecode as internal representation, which is a very analysis-friendly representation. No other code representation is required, therefore we do not need to translate between various representations. Secondly, the method cache allows us to calculate the instruction cache miss costs primarily based on the size of the methods.

Since Java does not allow indirect jumps or jumps out of methods and also keeps the class hierarchy information at the bytecode level, identifying methods and constructing control flow graphs and type hierarchy graphs is trivial compared to other binary formats. All analyses and optimizations are therefore performed on bytecode. Access to the source code is only needed for flow-fact annotations, there is no distinction between code transformations at source code level and binary level. Converting between different intermediate representations for interaction between the optimizer and the worst-case analysis is not necessary.

Because we use a method cache we only need to consider the code size of methods to calculate cache miss costs, instructions other than invoke and return instructions can be considered to be *always hit*. This simplifies the cache analysis. On conventional cache architectures, increasing the code size has a negative effect on the locality of the code and thus potentially leads to more cache line conflicts too, but the relation between method code size and cache miss costs is less explicit.

However, if the architecture uses a method cache, the instruction cache costs are proportional to the method code size. The gain of an optimization is therefore related to the code size increase that the optimization causes. Therefore we will first discuss the cache costs on the method cache. In the remainder of this chapter we then discuss several commonly used optimizations and their effects on JOP. At the end of Section 3.2 we will discuss the impact

of local improvements of the execution time on the worst-case execution time based on a simple example.

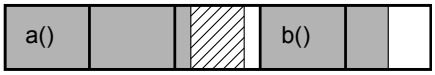
3.1 Code Size Considerations for the Method Cache

When a method cache is used, an optimization that increases the code size of a method will also increase the cache costs of invocations of that method, as well as cache costs of return instructions returning to the method that contains the inlined call site. Additional cache costs can appear, depending on the cache configuration and the precision of the cache analysis. In a variable block configuration (i.e., a method may occupy more than one cache block), a method may require more cache blocks after the code size has been increased.

Let N be the number of cache blocks of the method cache. Increasing the size of a method can cause some scopes that access at most N distinct cache blocks before the optimization to access more than N blocks after the optimization (see Section 2.3). This can cause some cache accesses not to be classified as *at most one miss* or *always hit* anymore, which increases the total cache costs. Other cache accesses may remain classified as *at most one miss*, but the scope for which they are classified gets smaller. So the upper bound for cache misses increases if the number of times that scope can be entered increases.

If the cache maps methods only to single cache blocks, cache accesses never need to be reclassified. Increasing the code size of a method only increases the invoke cache miss costs at the call sites of the method and the return cache miss costs of call sites within the method (if the return cache miss costs are attributed to the call sites instead of the return instructions). The cache cost increase linear with the code size, the speed of the program memory and the number of cache misses of accesses of that method, but the method sizes cannot increase beyond the size of the cache blocks.

For the variable-block method cache, which allows methods that are larger than a single cache block, the cache miss costs are also directly proportional to the code size, as long as the code size increase due to the optimization is small enough so that the optimized method requires the same number of cache blocks as the unoptimized version. An example is shown in Figure 3.1a. The code size increase of method `a()` represented by the shaded area does not affect the cache blocks for method `b()`. Also, if the cache analysis classifies all cache accesses as *always miss*, the cache costs increase linearly and only at methods that are adjacent to the optimized method in the call graph. In case of *always hit*, the cache costs are always zero. In all other cases there can be additional cache costs throughout the call graph as described above whenever a method requires additional blocks, although the total cache costs will always be below the cache costs estimated by an analysis that classifies all accesses



(a) The code size increase of `a()`, shaded area, does not affect `b()`

(b) Another increase of `a()` causes `b()` not to fit into the cache anymore

Figure 3.1: Effect of code size increases on cache allocations

as *always miss*. In the example in Figure 3.1b a second code size increase of method `a()` causes method `b()` not to fit into the cache together with method `a()` anymore. This results in additional cache misses when method `b()` is accessed after method `a()` has been loaded into the cache.

On the other hand, if the code size of a method is not increased by an optimization, the cache costs never increase. In case of JOP and the cache analysis we use, the total cache costs never increase if we reduce the code size, since individual cache miss costs never increase and the number of expected cache misses never increases for any call site if less cache blocks are accessed after optimization. We will therefore distinguish between optimizations that increase the code size of a method and optimizations that do not. In the latter case, if an optimization does not have a negative effect on the execution time of any other path in the control-flow graph of the method to optimize, the optimization can be always performed without any negative effect on either the ACET or the WCET, or the application code size. Feedback from the WCET analysis is not needed in this case.

If an optimization increases the code size of a method, the costs of all control-flow paths that contain a call site where the return can be a cache miss, as well as of all control-flow paths in invokers where an invoke of the optimized method can be a cache miss, can increase. This can cause the worst-case execution path to change.

3.2 Code Optimizations

There are a number of well known optimization techniques to reduce the expected execution time of some code, or to reduce the code size of the application. Although speed and code size are usually different optimization targets, a reduction in code size can also lead to an improved speed due to better cache behavior (however, this is not guaranteed on conventional architectures because of timing anomalies due to pipelining effects and potential instruction cache line conflicts).

Optimizations can aim at reducing redundancy in code (e.g., code motion, copy propagation or common subexpression elimination), at exploiting par-

allelism in the hardware (e.g., instruction scheduling or software pipelining), at replacing certain constructs with faster versions (e.g., strength reduction or constant folding), or at reducing the code size (e.g., dead code elimination). Many optimization techniques are presented in [1]. An overview of optimizations implemented in the WCC compiler can be found in [12].

In the following we will discuss some of the most common optimizations and their application to JOP.

- **Instruction scheduling and software pipelining**: Instruction scheduling rearranges the order of instructions in the code to minimize stall cycles due to resource conflicts or data dependencies of instructions, by scheduling independent operations between two conflicting operations. Software pipelining unrolls loops and rearranges the instructions in the loop body so that the execution of several iterations overlap. If the iterations of the loop are sufficiently independent of each other, the compiler can shuffle instructions from different, independent iterations between two operations of the same iteration.

  Since JOP waits during the execution of all bytecode instructions for all memory accesses to complete and does not share resources between instructions, those optimizations have no effect on this architecture.

- **Register allocation**: A typical compiler uses an unlimited number of virtual variables for its intermediate representation. To generate the code for the target architecture, the compiler must assign variables to registers on the target processor. If the processor does not have enough registers to store all variables in registers, the compiler must store some variables in memory which slows down execution. The objective of register allocation is to find a mapping that results in the lowest memory access costs.

  The JVM local variables can be regarded as a form of registers. The JVM allows a maximum of 2^{16} slots per method, but the maximum number on JOP is much smaller. Spilling local variables to memory however is not feasible. The first four slots can be accessed with special instructions that are shorter and, depending on the target architecture, are also slightly faster. Register allocation could be used to try to store the most commonly used variables in the first four slots, and to spill variables to auto-generated fields if required.

  However, since the difference between accessing the first four slots and any other slot is only one cycle on JOP, this has not been implemented. Optimizations are not performed if the resulting code would require more local variable slots than available on the target.

- **Strength reduction and peephole optimizations**: Strength reduction replaces certain instructions or instruction sequences with faster

versions. A typical example is a multiplication with a constant factor of 2^n. This can be replaced by a left shift by n bits, which is usually faster than a multiplication. On CISC architectures where a left-shift-and-add instruction is available, a multiplication by $2^n + 1$ can be replaced by a shift-and-add too.

Peephole optimizations look for specific code patterns that are typically generated by code generators and replace them with faster code sequences. Usually the new code is shorter or of equal length, and the gain is always positive. So there are no negative effects on the WCET in those cases.

- **Copy propagation and constant folding**: Copy propagation replaces usages of variables with their assigned value. This is used to eliminate unnecessary copy operations that are often created by other optimizations, and to propagate constants to the expressions where they are used.

  Constant folding replaces expressions by their value if the expression only uses statically known constants. This can cause edges in the control-flow graph to become infeasible, and thus allows the dead-code elimination to remove code that is statically disabled in the source code by constants. However, this is already done in the `javac` compiler, so this optimization may only have an effect after inlining. Since the optimization only removes instructions and changes the value of constants loaded onto the stack, such an optimization never increases the code size and thus has no negative effect on the WCET on JOP.

There are several ways in which loops can be transformed. A few of them are presented below. Most loop transformations need to transform the flow-facts of the loop, i.e., the loop bounds of the optimized loop need to be updated accordingly. Most loop optimizations change the order or the number of times the loop conditions are evaluated, so usually the conditions must be side-effect free for the optimizations to work.

- **Loop interchange, loop inversion and loop-invariant code motion**: Loop interchange swaps an inner loop with an outer loop. This can be used to iterate more efficiently over multidimensional arrays. Loop inversion changes a `while` loop into a `do/while` loop. If the compiler does not have a lower bound on the number of iterations of that loop, it needs to put the new loop inside an `if` construct that evaluates the loop condition. Code motion can be used to move side-effect free code that is independent of the loop iterations outside the loop body of a `do/while` loop, which reduces the number of executions of that code. Those optimizations have a low impact on the code size and the cache costs.

- **Loop peeling**: This optimization moves the first or last iterations of a loop out of the loop by copying the loop body. Loop condition checks can be omitted if the compiler has a lower bound for that loop. This can be used to remove special cases for some iterations from the loop, thus simplifying both the loop body as well as the code of the special iterations. A typical case is a loop that performs some initializations in the first iteration. By peeling the first iteration, the initialization code can be removed from the loop body, and the iteration check can be removed from both the initial iteration and the loop body. The code size increase and thus the cache costs depend on how many iterations are extracted and on how much initialization code can be removed from the loop body.

- **Loop unswitching and loop unrolling**: If a loop contains an `if-else` block with a loop-invariant condition, loop unswitching moves that condition to the outside by creating two copies of that loop inside an `if-else` construct with that condition, and replaces the conditional in the loops with their `if` or `else` blocks correspondingly. Loop unrolling duplicates the loop body, which reduces the number of times the condition is evaluated and the number of back jumps. Loop peeling must be used to ensure that the number of iterations is a multiple of the number of times the body has been unrolled.

  The disadvantage of those optimizations is that in case of loop unrolling the complete loop body is copied, possibly several times. This can increase the cache costs significantly for the method cache. Since branch delays are very short on JOP due to its short pipeline, the gain of those optimizations can easily be smaller than the additional cache costs, except for very small loop bodies. Lokuciejewski et al. show in [12] that for optimization sequences that target both WCET and code size, loop unrolling is not a good strategy, in contrast to the target pair (WCET,ACET) for a conventional instruction cache.

  Note that since the upper loop bounds must be known for all loops to perform a worst-case analysis, it is possible to unroll all iterations of all loops in the real-time code of an application, thus creating loop-free code that only contains loop condition checks for iterations between the lower and the upper loop bounds. This can simplify various code analyses, since the control-flow graphs are now directed acyclic graphs, and every operation of every loop iteration is represented at a different program location. Due to the large code size increase, however, this is not practical for code generation.

On the JVM architecture, we can also perform the following optimizations:

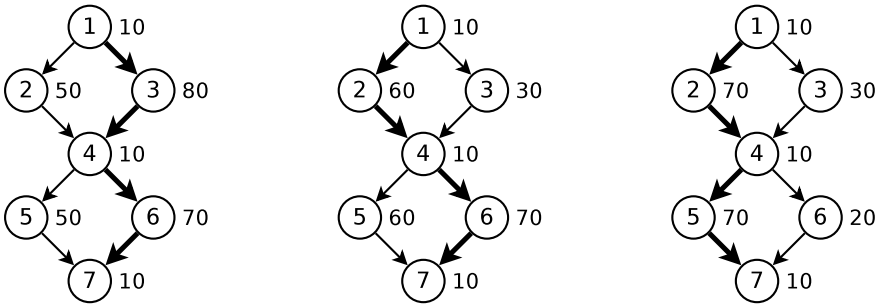
- **Stack code optimization**: The standard Java `javac` compiler stores every variable into a local variable slot. This is useful for debugging purposes, but temporary variables that are only used once or twice can be kept entirely on the stack in some cases, which reduces the number of required `load` and `store` instructions. Also, optimizations like inlining may introduce temporary variables. Peephole optimizations can be used to remove typical instruction sequences like `store/load` or `store/load/load`. More elaborate optimizations exist to optimize the instruction schedule and insert stack manipulation instructions so that fewer `load` and `store` instructions are required [14].

  The Soot framework uses two different techniques to generate stack code [32]. The first method is similar to the one described above. It generates straight-forward stack code from its internal representation, and then applies various optimizations to reduce the number of instructions. The second method first aggregates expressions from the internal quadruple code representation. Then efficient stack code is generated for those expressions using tree traversal techniques. This is similar to the method the `javac` compiler uses to generate stack code. Again, since this optimization only removes instructions or replace them with stack manipulation instructions, the code size should never increase. On JOP, rearranging the instruction schedule has no impact on the execution time since the execution time of bytecode sequences are composable, therefore this optimization should also never increase the WCET.

- **Reduce field accesses**: If a non-volatile field of an object is accessed multiple times in a method, some of the accesses can be replaced by much faster accesses to local variables by copying the field to a local variable at its first usage and writing the variable's value back after its last assignment. This can reduce the execution time of a method with little or no code size increase, especially after inlining has been performed.

Most standard optimizations do not increase the code size significantly and therefore have a low impact on cache costs. In [12] the authors also show that standard optimizations have a similar impact on WCET and ACET. On JOP, instruction-reordering optimizations are not required. Due to the method cache, however, we need to avoid code size increases even more than on conventional architectures.

In combination with a WCET analysis, another objective of code optimizations can be to create code that is better suited for analysis, e.g., by peeling the first loop iteration so that cache accesses within the loop body for all other iterations can be classified as *always hit*, or by inserting instructions to reset the processor state or clear the cache, as discussed in Section 2.3.



(a) Initial control-flow graph, WCET is 180

(b) CFG after optimization of node 3, WCET is 160

(c) CFG after optimization of node 6, WCET is 170

Figure 3.2: Example of WCET path switching during optimization

Rolling back the code transformations done by an optimization is called deoptimization. This can be used to undo an optimization when the WCET analysis shows that it had a negative effect on the WCET.

Optimization Gain and WCET Improvement

So far we have only considered the gain of an optimization for a single execution of the optimized code path. However the actual WCET decrease can be much lower. If the optimized code is not on the WCEP, the WCET does not decrease at all.

Figure 3.2a shows a simple control-flow graph. The nodes represent basic blocks, the labels next to the nodes represent the execution time of the blocks. The worst-case path is shown by bold edges. We assume that nodes 2 and 5 contain call sites where the return is a cache miss.

We assume that we can reduce the costs of nodes 3 and 6 by 50, the costs of node 5 by 40 and of node 2 by 20 (including cache cost increases), and that optimization of any node increases the method code size so that the cache miss return costs increase by 10. Furthermore we assume that we can only optimize two nodes because any additional optimization would increase the code size of the method beyond the maximum code size limit. The cost changes for optimizing a single node are summarized in Table 3.1.

We choose node 3 as the first node to optimize. Although we reduced the execution time by 50, the WCET only decreases by 20, because the worst-case path now switches to node 2, and we additionally increased the costs of node 2 and 5 due to the increased code size. Figure 3.2b shows the control-flow graph with the new WCEP and the new execution costs. If we would have chosen node 6 instead, the WCET gain would only be 10, although the optimization gain at node 3 and 6 is the same.

We now have two possibilities. We can either reduce the costs of node 2

	Node 2	Node 3	Node 5	Node 6
Initial	50	80	50	70
Optimization of 2	-20	0	+10	0
Optimization of 3	+10	-50	+10	0
Optimization of 5	+10	0	-40	0
Optimization of 6	+10	0	+10	-50

Table 3.1: Cost changes for optimizing a single node in Figure 3.2a

by 20 or of node 6 by 50. A greedy algorithm would certainly choose node 6 in this case, if it only considers the local gain. However this is a bad choice in this case, leading to the new costs shown in Figure 3.2c. The WCEP switches again to node 5, making the gain of the optimization void, and the additional cache costs at node 2 even increase the execution time of the WCEP. Optimizing at node 2 would have further reduced the WCET to 140, the higher cache costs at node 5 do not matter. Note that the total cache costs only increase by 20 while we reduce the costs of node 6 by 50, still the WCET increases by 10.

This however is still not an optimal solution for our example. If we would have chosen to optimize node 6 instead of node 3 at the beginning, the WCET would initially decrease only by 10, and the new WCEP would go over nodes 3 and 5. But if we would then optimize node 5 and reduce its execution time by 40 (although we could reduce the costs of node 3 by 50), the new WCET would only be 130.

The WCET reduction therefore depends not only on the execution time reduction of nodes on the WCEP, but also on the execution time of alternative paths, the influence of WCEP optimizations on those other paths, and if those paths can be optimized too.

As we have seen, the WCET decrease for a method can be lower than the local optimization gain. The same holds true for the global WCET. If we decrease the WCET of a method, the costs of all call sites of that method are reduced by the same amount (assuming that the call sites may not call other methods). The WCET decrease of the invoker method however will be even lower than the WCET decrease of the invoked method if the WCEP switches away from the call site in the invoker. The global WCET gain of an optimization therefore depends not only on the execution time of alternative paths bypassing the optimized code, but also on the execution time of paths bypassing the call sites on the global WCEP that can (indirectly) invoke the optimized method.

On the other hand, an increase of the execution time of code outside the WCEP is at least partially hidden, although it can reduce the WCET gain of other optimizations. In our example, the cache cost increases at node 5 do not increase the WCET at all, but they reduce the WCET gain of the

optimization of node 6.

3.3 Function Inlining, Cloning, and Splitting

If it is possible to statically determine that a call site always invokes the same method, it is possible to replace the call site with the code of the invoked method. This optimization is called function inlining. It reduces the call overhead and can create additional optimization opportunities, since the invoked code can now be optimized in the context of the call site. On the other hand, the code size increases in the general case, which usually also increases cache costs (see Section 3.1).

Function cloning or specialization also enables additional optimization opportunities by creating a separate version of a method for a specific call context. The new version can now be optimized specifically for this context. This does not reduce the call overhead (except if constant arguments are propagated into and removed from the new method) and can yield fewer optimization opportunities than inlining as the code cannot be optimized across method boundaries. However it also does not increase the code size of the invoker method. The cache costs might still increase since the optimized method may now appear in the cache more than once, which can lead to additional cache conflicts. However, only the number of cache misses may increase, not the cache miss costs of any cache access.

Function splitting can be seen as the inverse operation to function inlining. It replaces a subgraph of a control-flow graph with a call to a new method containing the replaced code. Although this adds an additional call overhead, the size of the optimized method is reduced. This can reduce the cache miss costs, and it can be used to reduce the size of methods that are larger than the maximum method size allowed by the method cache. It should be noted though that this is not the same as deoptimization of an inlined call site, which creates a call to an existing method, while function splitting creates a call to a new method, which has a different impact on the cache.

Reducing the call overhead by function inlining does not lead to the same reduction of the WCET, since the WCEP can change and thus hide part of the gain. Similarly, the additional call overhead required for function splitting does not lead to an increase of the WCET by the call overhead, if the extracted subgraph is not part of the WCEP. If the execution time of the worst-case path is higher than the execution time of the optimized path plus the call overhead, the WCET does not increase at all. The actual WCET improvement of inlining and function splitting heavily depends on how the cache miss cost changes affect the worst-case execution path.

Inlining is the main optimization employed in this thesis. The main reason for this choice is that invocation costs are very high on JOP, thus the potential gain of removing invocations is also high. Since inlining not only changes the

optimized method but also the call graph, can introduce new call sites, affects context references, and can lead to some methods becoming unused, it has an impact on virtually all data structures in the optimizer.

Two separate inlining optimizations were implemented. The first inliner, called SimpleInliner, is designed to inline without increasing the code size. This optimization is very fast and does not require interaction with the worst-case analysis. The second inliner handles all other cases and can increase cache costs. The greedy algorithm presented in Section 4.4 is used to choose call sites for optimization for which inlining is expected to improve the WCET. More details on the implementation of the inlining optimizations and how the data structures and analyses are updated can be found in Chapter 4.

3.4 Code Size Reduction

To reduce the code size of the final application, unused code can be removed from the application. The current optimizer first removes all unused members (classes, fields and methods) and debug annotations. Then the constant pools of all classes are reconstructed using only used entries. Those two optimizations are presented in more detail in Section 5.4. They only affect the application size and do not change the execution time of the application.[1]

On the other hand, a dead code elimination reduces the size of methods, which has a positive effect on cache costs. The loop bound data-flow analysis can also detect infeasible edges in the control-flow graph. This can be used to find all basic blocks in a method that are only reachable over infeasible edges and can therefore be removed. However, removing dead code is currently not implemented. Feedback from the WCET analysis is not required for this optimization.

The `javac` compiler does not emit code for statements that are enclosed by a condition that statically evaluates to false. The data-flow analysis may find additional infeasible edges, although it could be argued that code that is unreachable due to conditions less obvious than simple expressions over constants can be seen as a hint to an erroneous implementation.

[1] It is possible that due to smaller constant pools, the shorter version of `ldc` can be used at some places, which can reduce the method size and thus the cache miss costs of that method, but that effect is negligible. An increase of cache costs or the execution time due to the optimizations is avoided by design of the optimizations.

Chapter 4

Method Inlining for Java

As described in the previous chapter, inlining is used to reduce the invocation overhead at some of the call sites in the application. The drawback of inlining is that the code size is increased, which may increase the cache miss costs significantly. On architectures using the method cache, the code size increase translates directly into increased cache miss costs at several call sites. For other cache architectures, inlining can cause an increase in cache line conflicts. Inlining also increases the register pressure in the invoker. Additional spill code may be required when a call site is inlined. The JVM provides a large number of local variables, so this may not be an issue. However the target architecture can limit the number of available local variables per method significantly. JOP currently allows at most 32 local variables per method.

Besides changing the execution time of the optimized code, inlining also eliminates a call site, creates new call sites in the invoker if the invoked method contains call sites, and has an impact on the cache classification at several locations in the application. Hence the cache analysis, the worst-case analysis and the call graph must be updated accordingly.

In this chapter the implemented bytecode inliner is presented and we show how the analysis results are updated during inlining. The greedy algorithm presented in Section 4.4 selects candidates for inlining. At the end of the chapter other approaches for selecting call sites are briefly discussed.

4.1 Preparations

To perform inlining, we first need to determine the receivers of virtual invokes. Then all call sites for which the invoked method implementation can be determined statically are analyzed to check if inlining is possible and allowed by the configuration.

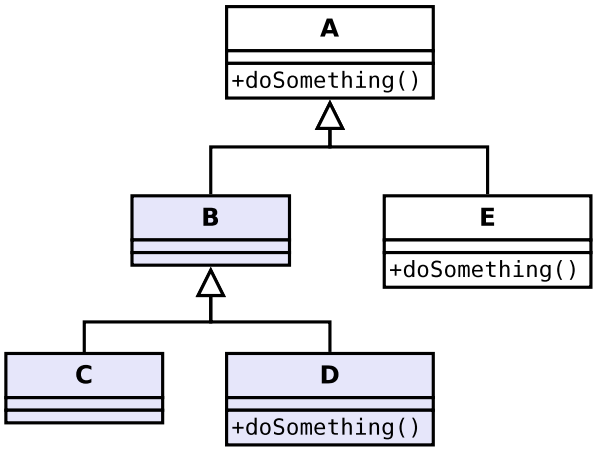


Figure 4.1: Example class hierarchy. A virtual invoke referring to `B.doSomething()` may invoke implementations from classes `A` or `D`

Resolving Virtual Invokes

Non-virtual invokes always execute the method they refer to (either because they refer to methods that cannot be overloaded like static or private methods, or because a specific method implementation should be invoked, e.g., a super-method). Virtual invokes however may invoke a number of different methods.

The set of methods that can be invoked at a call site contains the method referenced by the invoke instruction if the declared type can be a receiver type, and all method that override the referenced method defined in any receiver type. To be able to inline a call site, this set must contain exactly one method. Finding the implementing methods of a call site is done by using either a class hierarchy analysis or the receiver type data-flow analysis (which is also used by the worst-case analysis). If a call graph has already been constructed as described in Section 2.4, it can also be used. The successors of the nodes representing the call site in the call graph are the method implementations that can be invoked.

For example, if we have a class hierarchy as shown in Figure 4.1 and a virtual invoke calls `B.doSomething()`, then the implementation from either class `A` or `D` could be invoked, depending on the receiver of the invoke instruction. `E.doSomething()` can never be invoked in this case, because although it overrides a possible implementation, class `E` cannot be a receiver of this invoke instruction. If we could prove either that class `D` (that provides its own implementation of `B.doSomething()`), or that both classes `B` and `C` (that inherit `A.doSomething()`) are never receivers of the invoke, the invoked implementation is statically known and we can inline the call site.

So far we assumed that all classes of the application are known to the compiler and that therefore the class hierarchy is complete. If not all classes

referenced somewhere in the used application code are loaded or if reflection is used to instantiate classes, the class hierarchy might contain additional subclasses and thus potentially additional receivers at runtime, which could provide additional implementations. In this case inlining is only safe if the method in the declared type of the invoke is final or if the DFA can prove that there are only known receivers. The super classes must also be known if the implementing method is inherited (class `A` in our example, if the set of receivers contains class `B` or `C`).

For applications that are executed on JOP it can be assumed that the whole program code is available and that reflection is not used. This can also be checked by the compiler. Desktop JVMs, that cannot make those assumptions, can instead insert code before the inlined code that checks that the actual receiver type matches the type that has been used for inlining and falls back to a virtual invoke if this is not the case, but this degrades the performance of inlining. This method can also be used to decrease the average-case execution time by inlining the most commonly invoked method, but this increases the WCET so it is not an option for real-time applications.

Dynamic class loading also makes inlining unsafe, since the code of the implementing method can be changed at runtime. In this case inlining would only be safe if all inlined invoke sites would be updated or deoptimized if a method is replaced with a new version at runtime. Again, this is not possible on JOP so the inliner does not need to handle this case.

Inlining Checks

Even if the implementing method can be determined statically, inlining the method might not be possible for several reasons. Conditions that must be satisfied so that a method can be inlined successfully are presented in [3] and [30]. The conditions are briefly presented here:

- **Native methods**: Native Java methods cannot be inlined and are therefore ignored.
- **Recursive methods**: Recursive calls are not inlined to prevent the inliner from getting stuck at unbounded recursive calls. The maximum method size is limited so the inliner would abort anyway if the method size increases due to inlining, but this does not need to be the case if the program to optimize contains bugs (i.e., if it contains a method that only invokes itself).
- **Maximum code size**: The invoked method cannot be inlined if the resulting invoker is larger than the maximum code size for methods, which can be restricted e.g., by the method cache.
- **Number of local variables and stack size**: To inline a method additional local variables may be needed to store the local variables of

the invoked method. The number of available local variable slots can be limited by the target processor. Inlining is not performed if there are not enough slots available. It would be possible to generate spill code at the call site to store live local variables into auto-generated fields, but the costs may be higher than the cost of a call. A similar restriction holds for the stack size although this is much less likely to prevent a call site from optimization and the stack could be saved to local variables if required. The stack contents at the call site needs to be stored in local variables in any case if the invoked method contains exception handlers, since catching an exception clears the stack contents. If the invoked method catches an exception, the invoker would not be affected in the unoptimized version since only the stack of the callee is cleared, but after inlining the stack of the invoker would be cleared too.

- **Excluded methods**: Methods can be excluded from being inlined by configuration. The method for which the WCET should be calculated must not be inlined, otherwise the executed code would be different from the code analyzed by the WCA. It can also be desirable not to inline into instrumented methods that are used to measure the execution time of a method including initial invocation costs.

- **Access checks**: If a method is inlined into another method in a different class, all non-public fields and methods accessed by the method need to be changed to package or public access. If a method is changed to public access, all methods with the same signature in all subclasses need to be made public too.

  It is possible that package visible methods are overridden by methods in subclasses in other packages due to the access modifications. In this case the inliner can only optimize the call site if either the invoked method and its overriding methods or the non-overriding methods in the subclasses are renamed and the call sites are updated to refer to the renamed methods correspondingly.

  If the call site tries to invoke a method that is not accessible to the invoker, the call site is not optimized. This can happen if the class containing the callee is recompiled but the class containing the call site is kept. Such code will throw an exception at runtime if this access violation was not reported already at compile time.

- **Special invokes in the callee**: Additional checks are required if the callee contains `invokespecial` instructions. If `invokespecial` refers to a superclass of the class of the callee, the `ACC_SUPER` flag is set and the invoked method is not a instance initialization method, the instruction does not call the referenced method, but calls the method with the same signature defined in (or inherited by) the direct superclass of the class

```
class B {            class C extends A {        class A {
  void foo() {         void m() {                 void m() {
    invoke C.m()         invokespecial A.m()      }
  }                    }                        }
}                    }
```

Figure 4.2: An `invokespecial` instruction in the callee

that contains the `invokespecial` instruction (also see Section 2.2). If the `ACC_SUPER` flag is set and the instruction does not call an instance initialization method, we need to check if the inlined code still calls the same method.

Let class `A` be the class of the method invoked by the `invokespecial` instruction in the callee and let `B` be the class of the invoker, as shown in the example in Figure 4.2. In the example, we want to inline the call site in `B.foo()`. We assume that after the optimization the class hierarchy is not changed and no new methods are added to the application. Then we only need to check if the invoke instruction in the callee resolves to the same method as the inlined call in the invoker.

The inlined invoke instruction will call the same method as the original `invokespecial` instruction if `B` is not a subclass of `A` or if `B` is a subclass of `A` but there is no class `C` that is a subclass of `A` and a superclass of `B` and contains a method that overrides the method referenced by the invoke instruction. In the first case, the inlined `invokespecial` in the invoker is not a super call and will thus always call the referenced method. In the second case the inlined invoke is a super call since `B` extends `A`, but the invoke resolves to the same method as in inlined callee.

Otherwise we can only inline if we clear the `ACC_SUPER` flag of the invoker class. In our example this would be the case if `B` extends `C`. Then the invoke in `C.m()` resolves to `A.m()`, but when we inline `C.m()` into `B.foo()`, the `invokespecial A.m()` in `B.foo()` would resolve to `C.m()`.

We do not even require that we inline a method from a super class. Assume that in our example we have a fourth class `D` that has a method `bar()` containing `invokespecial A.m()`, and let `B.foo()` invoke `D.bar()` and let `B` extend `C`. If we then inline the call site in `B`, the special invoke from class `D` would again invoke `C.m()` instead of `A.m()`.

Note that if `C` contains a package visible method with the same signature as the method in class `A` that is invoked by `invokespecial`, but does not override the method in class `A`, the access checks will prevent the method in `C` from being changed to public access.

Non-static invoke instructions are required to throw null pointer exceptions if the receiver is a null reference. A null pointer check needs to be inserted in

the prologue of the inlined code to emulate this behavior. If the optimizer can verify that the inlined code will always throw a null pointer exception before any side effects are produced, the null pointer check code can be omitted. This is done by checking the first unconditionally executed instructions of the callee for instructions like `getfield`, `setfield` or invoke instructions. If they use the `this` reference as receiver, they throw the same exception as the optimized invoke if the `this` reference is a null pointer. The data-flow analysis could be used to check if the receiver of a call site can be a null reference, but this has not been implemented in the analysis yet.

Java Implementations of Bytecode Instructions

Bytecode instructions that are implemented as Java methods in the JOP JVM can be seen as special static invokes. They can be inlined in the same way as `invokestatic` instructions. The only difference is that the return type and argument types of the invoked method do not need to match the types that are expected on the stack. This is used for instance by the floating point software implementation, which provides implementations for bytecode instructions that operate on float and double types on the stack. The implementation however accesses the values as int and long types to perform bit manipulations. Although the methods could be inlined, the resulting code would not be legal bytecode and would not be accepted by a bytecode verifier. Such cases are therefore not inlined by default, however this check can be disabled by an option of the optimizer.

4.2 The SimpleInliner

If the code size is not increased when a call site is inlined, there is no increase in cache miss costs or in the global code size. The execution times of other paths are not affected, therefore inlining has no negative effects on the WCET or the ACET for such call sites.

A specialized method inliner called SimpleInliner has been implemented to optimize call sites that can be inlined without an increase in code size. The inliner only optimizes call sites where no or only a very short prologue for parameter passing needs to be inserted. By restricting the type and the sequence of instructions in the callee, SimpleInliner can generate parameter passing code more efficiently than the generic inliner described in section 4.3.

As long as the code size is not increased, the gain of the optimization is always positive, therefore this optimization does not require interaction with the WCET analysis or the method cache analysis. The order in which the call sites are processed is not important either. If a method `b()` can be inlined by the SimpleInliner into a method `a()`, and if `b()` contains a call site that can be inlined by SimpleInliner too, the call site in `a()` can be inlined using either the optimized or the unoptimized method `b()`. Therefore the optimization

can be applied to all methods of the application and to all call sites in the methods in arbitrary order. New call sites in the inlined code are optimized recursively.

The SimpleInliner is used to inline getter, setter, and most wrapper methods, as well as method stubs and methods performing simple calculations and empty methods, including initialization methods. The optimization is very fast and can be applied to optimize at least some invocations if the developer does not want to use the slower WCET driven optimizations. It also reduces the number of call sites for the more complex generic inliner.

Selecting Call Sites

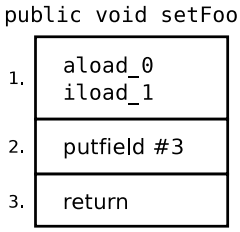
First, the inliner performs the actions described in Section 4.1 to resolve virtual invokes, to check if inlining is allowed and to change the access of members if required. Inlining is not performed if the callee contains exception handlers, requires an additional null pointer check, or if the callee code size is larger than a certain threshold, as in those cases the code size of the invoker would most certainly increase. A precise code size check is performed after the new prologue for the call site has been generated.

The maximum size of a callee is equal to the size of an invoke instruction (5 bytes for `invokeinterface`), a return instruction (1 byte), plus the size of instructions used for parameter passing in the callee and at the call site, if they can be removed from the original code as described below.

Inlining Selected Call Sites

The SimpleInliner inlines methods if the code matches the following layout:

1. Instructions that load parameters or constants on the stack
2. Instructions that perform operations on the stack, e.g., arithmetic operations, invoke instructions, field access, stack manipulations (like `dup` or `pop`), type checks or constant loads.
3. A return or throw instruction



A method matching the required layout

Branch instructions and load and store instructions are not allowed in the second section. It would be possible to allow load and store instructions that use slots different from the slots containing the arguments, and to allow branch instructions that do not jump into the first section, but such methods would most likely be too large for inlining anyway.

```
void invoker():
  ...
11: getfield <myObject>
14: iload 4
15: iconst_1
16: invokeinterface
     <MyIF.calc(II)I>
21: istore_3
 ...
```

```
int calc(int a, int b):
  // return this.calc(b,a,null);
  0: aload_0
  1: iload_2
  2: iload_1
  3: aconst_null
  4: invokespecial
      <MyObj.calc(IILOtherObj;)I>
  7: ireturn
```

(a) Original code

```
void invoker():
  ...
11: getfield <myObject>
14: iconst_1
15: iload 4
16: aconst_null
17: invokespecial
      <MyObj.calc(IILOtherObj;)I>
20: istore_3
 ...
```

(b) Optimized code

Figure 4.3: Example of a call site optimized by SimpleInliner

The inliner first analyzes the stack state after the first section, mapping each stack entry to the method argument or constant value it contains. Then the optimizer tries to modify the code at the call site so that the top of the stack at the call site contains the same values. If the optimizer succeeds, it replaces the invoke instruction with the second code section. Note that any of the first two sections can be empty.

All instructions that load local variables, constants or the value of a static field onto the stack at the call site are rearranged, as long as they are within the same basic block. If the callee consumes more values than those loaded by those instructions, the callee must use them in the same order as they appear on the stack (i.e., the method must start with loading the arguments in the same order as they appear in the method descriptor).

Figure 4.3 shows an example of a call site in method `invoke()` that gets optimized by the inliner (the numbers in front of the instructions represent the byte address in the code). The invoker calls `this.myObject.calc(x,1)`. We assume the call resolves to the given `calc(int,int)` method, which in turn calls a private `calc(int,int,OtherObj)` method.

The instructions at address 14-15 are updated using the stack value map-

ping calculated from instructions 1-3 in the callee. The `this` reference of the callee is placed on the stack using a `getfield` instruction at the call site, which is not rearranged by the optimizer. However since the callee consumes the `this` reference exactly once and places it at the bottom of its stack with the `aload_0` instruction at address 0, the call site does not need to be modified for this parameter. Finally the original call site is replaced with the rest of the code of the callee, namely the invoke instruction at position 4.

If the invoker and the method invoked by `invokespecial` are in different classes, the inliner must make the private method `calc(int,int,OtherObj)` public. The new call site at position 17 in the invoker is a new optimization candidate, so the optimizer then starts all over using the new call site, if the new callee is small enough. Since the callee method is different from the previously inlined method, it is not a recursive call.

4.3 The Inline Optimizer

While SimpleInliner is designed to handle very small methods that match a given pattern, the inliner described in this section can inline any method as long as the inlining criteria presented in Section 4.1 are met. The code generation for parameter passing is less elaborate, removing unnecessary local variable copy operations is left to other optimizations.

Inlining all call sites could lead to an unacceptable increase of the code size and the cache costs. Therefore the greedy algorithm presented in Section 4.4 is used to select only a subset of the call sites for inlining to minimize the negative effects of inlining.

The inliner must also update analysis results in accordance with the optimization, so that the selection algorithm does not use outdated results for its decisions during optimization. The updates performed during inlining are presented in Section 4.5.

Selecting Call Sites

The inliner first performs the inline checks described in Section 4.1 for every call site in the methods to optimize. If the receiver of a call site can be resolved uniquely and if inlining is allowed, the optimizer calculates the expected gain, the code size increase and the expected cache miss cost increase as described in the next two sections, and passes this information along with the call site to the greedy algorithm described in Section 4.4.

The greedy algorithm checks the code size constraints and selects call sites for which it estimates the best gain to code size ratio, which are then optimized by this inliner.

Code Size Changes and Gain

The code size increase of the invoker method by inlining a call site is equal to the size of the callee, plus the size of the inserted prologue, minus the size of the replaced invoke instruction. The size of the inlined code can be larger than the size of the callee, because the local variables used by the inlined code will be mapped to higher slots than in the callee, therefore larger parametric instructions like `iload` may need to be used instead of smaller instructions like `iload_0`. While the last return instruction can be removed, other return instructions need to be replaced with larger `goto` instructions.

The application code size increases by the same amount, except if the call site to inline is the last call site of the callee in the application. In this case the callee is not used anymore and can be removed. Then the application code size increase only includes the size of the prologue and the increase of the callee code, not the size of the callee code itself. Note that even if a method is not used anymore after inlining, its invoked methods will still be used by the inlined code. Therefore we do not need to check if any method other than the callee becomes unreachable in the call graph.

Similarly, if we ignore the cache costs, the execution time gain of the optimization for a single execution of the call site is equal to the execution time of the removed invoke instruction (assuming a cache hit), plus the execution time difference between a single return instruction and a goto instruction. The gain is reduced by the execution time of the prologue (we can assume that the receiver is never null for the execution time of the null pointer check). The gain can be further decreased since the parametric versions of instructions accessing local variables may have a higher execution time, but this effect is ignored by the current implemention of the gain estimator.

Impact on Cache Miss Costs

The application wide cache cost changes for inlining a single call site can be expressed as the sum of four values. The number of cache misses are provided by the cache analysis used by the greedy algorithm and are relative to a single execution of the target method (i.e., the root method of the call graph containing the methods to optimize).

1. Since the invoke instruction is removed, the total cache costs are decreased by the invoke and return cache miss costs that appear at the inlined call site.

2. All call sites in the callee are duplicated and copied into the invoker. Since all of those call sites still call the same methods, the invoke cache miss costs of those call sites do not change. However, since the inlined call sites now return to the invoker that now consists of the code of the

callee plus the old invoker, the return cache miss costs of the inlined call sites are increased proportionally to the size of the invoker.

If the number of cache misses in the callee decreases by the same amount of cache misses that now occur in the inlined code (i.e., if the total number of cache misses stays constant), the total cache costs are increased by the product of the return cache miss cost increase and the number of return cache misses that now occur at call sites in the inlined code.

Otherwise, if returning to the invoker causes more cache misses than returning to the original callee, the cache miss cost increase can be calculated as the total cache miss costs of the inlined code being executed n times, minus the decrease of cache miss costs due to the callee being executed n times less often, where n is the execution frequency of the optimized call size.

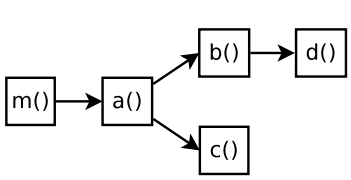
3. The cache costs are increased further due to the code size increase of the invoker, because all invoke cache misses of the invoker and all return cache misses of all call sites in the invoker have higher cache miss costs, proportional to the size of the callee (except for the call sites in the inlined code, they have already been handled above).

4. Furthermore, the total cache costs of the application can increase if the invoker requires additional cache blocks after the optimization, as discussed in Section 3.1. On the other hand, if the inlined callee is no longer reachable from the optimized invoker, and if the callee is small enough so that the invoker does not require additional blocks, then the number of cache blocks required by all methods reachable by the invoker is reduced by one, which may decrease the number of method cache misses.

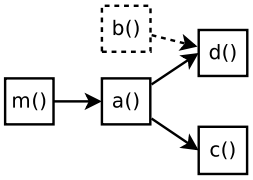
   However, the estimated cache miss cost change caused by the change of the number of required cache blocks depends on the cache approximation used by the cache analysis.

The total gain of optimizing a call site including cache costs can be estimated as the gain without cache costs per execution of the call site multiplied by the execution frequency of the call site in the application, minus the cache cost difference as described above. The actual WCET gain however can differ significantly, since those calculations do not honor the fact that the worst-case path can change when the cache costs change, as discussed in the example in Section 3.2.

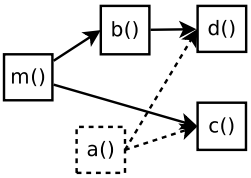
Since the cache costs, and hence the total gain, depend on the size of the invoker and the callee as well as on the estimated number of cache misses, the gain estimations for all call sites of a method m must be updated if a call site in method m or any method invoked by m is inlined, or if the cache analysis results change for method m.



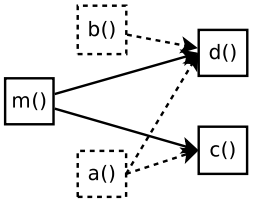
(a) Initial call graph



(b) Method `b()` has been inlined into `a()`



(c) Original code of `a()` has been inlined into `i()`



(d) New call site of `b()` has been inlined

Figure 4.4: Inlining using unoptimized method code

Inlining Unoptimized Methods

The current implementation of the inliner does not keep separate versions of the optimized and the original method code. When a method is inlined, the inlined method may already contain inlined code. Although this has the advantage that the implementation of the inliner is simpler since we do not need to keep separate versions of methods and the corresponding analysis results, it has the disadvantage that the order in which call sites are selected has an impact on which call sites can be optimized. If the size of the callee is increased too much due to inlining, it may not be possible to inline a call site anymore due to code size restrictions.

If the inliner would keep the original unoptimized code of all methods and would always inline unoptimized code, optimizations of the callee of a call site first would not affect the code size increase of the invoker, so inlining does not become impossible during optimization. Also, if unoptimized code is inlined, the optimizer can choose not to inline a call site in the inlined code although it has already been inlined in the callee. This allows the optimizer to find better trade-offs between code size and speed for each method, and to optimize the code differently in different contexts.

If a method `a()` invokes a method `b()` as shown in Figure 4.4a, and all call sites of `b()` get inlined (including the one in `a()`), then method `b()` is not used anymore and can be removed from the application code, and does not allocate method cache blocks (Figure 4.4b). But if a call site of `a()` is inlined, then the new code will again contain a call site of method `b()` (Figure 4.4c). If the inliner does not inline the new call site too (e.g., because of method size restrictions) the method `b()` cannot be removed anymore.

This has several consequences. First, when a call site is inlined, the application code size can increase by more than just the size of the callee, since methods that have already been removed are used again, thus possibly making the advantage of inlining the last call site of those methods void. Second, the gain (ignoring cache costs) is lower since we replace an invoke of a possibly already optimized method with the unoptimized code of the method. This also means that analysis results like execution frequencies and worst-case execution time need to be kept and updated for both the unoptimized versions of methods as well as unused methods, as they are needed to calculate the effects of the optimization. However if the unoptimized inlined code is optimized in the same way as the callee, the gain and the code size changes are the same as when the optimized code is inlined (Figure 4.4d).

In both cases inlining has an impact on the application code size and can change the WCET and WCEP not only on all paths from the root to the optimized method in the call graph, but also in other regions of the call graph, depending on the cache analysis. Since the gain of inlining a call site depends mostly on the cache costs, inlining of one call site can make inlining at other call sites infeasible.

Inlining Selected Call Sites

Once a call site has been selected for optimization, inlining is performed as described in [3]. A prologue is created that pops all parameters of the invoked method from the stack and stores them into unused local variable slots. A null pointer check for the `this` reference of non-static callees is inserted if required.

The invoke instruction is replaced with the code of the invoked method. Local variable accesses are mapped to the slots where the parameters have been stored by the prologue or other unused slots. Return instructions are replaced with jumps to the instruction following the inlined invoke instruction. If the stack prior to a return instruction in the callee contains more than the return value, `pop` instructions are inserted accordingly to create the correct stack size after the return.

Loop bounds and source line numbers are copied from the callee. If the class of the inlined method differs from the class of the invoker, a reference to the source class is attached to the line numbers. Since loop bounds are relative to their enclosing loop in our implementation, the loop bounds do not need to be adapted if the call site appears within a loop. However, if context sensitive loop bounds would be used (i.e., loop bounds that can be more precise in certain contexts), they need to be updated to the new context.

4.4 WCA-driven Greedy Optimizer

In this section we present a WCA-driven greedy algorithm to select optimization candidates iteratively. The basic idea behind this algorithm is similar to

the algorithm presented in [37], but this algorithm can be used for other optimizations than inlining too, and it uses a worst-case analysis for the selection of candidates. It is also possible to select between candidates of different optimizations, eliminating the need for a phase ordering of those optimizations. A disadvantage of that algorithm is that it does not honor the fact that the WCET gain of a set of optimized candidates can be different from the sum of the gains of the individual candidates.

The inliner provides a set of optimization candidates to the algorithm. For every candidate the algorithm requires a code range containing the code to optimize, the code size increase of the optimized method, and the estimated execution time gain for a single execution of the optimized code excluding cache costs, as well as the total cache miss cost increase for instructions in the modified code range. In addition to that, the algorithm needs to know which methods are no longer invoked by the optimized method, and which methods are no longer reachable by the entry method in the call graph (i.e., which are no longer used).

An execution frequency analysis and a method cache analysis are used to estimate the gain and the cache costs. The worst-case analysis is used to decide which candidates should be selected. The analyses and the interaction with the WCA tool are presented in the following sections.

To favor candidates that have a low impact on the application code size, we use the ratio of the expected gain to the code size increase to select candidates. The algorithm works by repeatedly selecting the candidate with the highest rebate ratio, i.e., the highest estimated gain per code size increase. It optimizes he selected candidates6 and updates the call graph, the execution frequency analysis and the method cache analysis. Then the ratios of the candidates are reevaluated. The algorithm stops when either no candidate with a positive gain is left or until the application code size reaches a predefined threshold. The code size is therefore not an optimization objective, but a constraint. However since the cache costs depend on the code size, the optimizer will favor optimizations with low impact on the code size due to the cache costs too.

The pseudo-code of the optimizer is shown in Algorithm 4.1. In more detail, the algorithm performs the following steps:

1. **Initialization**: First, the method cache analysis and the execution frequency analysis calculate the initial analysis results. The WCA tool is used to calculate an initial worst-case execution path.

2. **Iterate over call graph regions**: Depending on the configuration, the methods to optimize in the next steps are chosen. We can either choose to optimize all methods reachable from the entry method together (i.e., we choose the best candidate from all candidates in all methods), or we can choose to optimize only one method at a time (i.e., choosing the best candidate within a single method) and traverse the call graph in

Algorithm 4.1: Pseudo-code for greedy optimizer

```
initialize WCA, cache analysis, exec frequency analysis;
foreach region in call graph do
    find initial candidates in region;
    calculate ratios for all candidates;
    while candidates available do
        select candidate c with highest ratio;
        optimize c;
        find new candidates;
        update analyses;
        update ratios;
    end
end
```

a top-down or bottom-up manner. By default we optimize all methods reachable from the WCA target method first, and optimize the rest of the application afterwards.

3. **Find initial candidates**: The selected call graph region is searched for optimization candidates. The found candidates are evaluated and sorted using the calculated rebate ratio. There are several ways to evaluate and classify the candidates; this is discussed at the end of this chapter.

4. **Select and optimize a candidate**: The candidate with the highest rebate ratio is optimized. The optimization must also update the execution frequency analysis and the method cache analysis, as well as data structures like the call graph accordingly, since inlining creates new execution contexts, creates new edges in the call graph and changes the method code size. Inlining and updating the analyses is performed by the inline optimizer as discussed in Section 4.3.

5. **Update candidates**: Optimizing code can create new optimization opportunities. We therefore search the optimized code region for new candidates. We also need to reevaluate the rebate ratio of all candidates where the optimization gain, the cache miss costs, the WCEP, or the code size changed.

6. **Continue optimization**: After the candidates have been updated, we continue at step 4 if we have any candidates left. Otherwise we go to step 2 and select the next call graph region to optimize or stop if no more methods are left to optimize.

Execution Frequency Analysis

We can approximate the execution frequency of an instruction for one execution of the method containing the instruction by using the upper loop bound for instructions inside a loop, and assume an execution frequency of 1 for all other instructions. The result is an upper bound on the real execution frequencies. The estimation of the gain of an optimization (excluding cache costs) based on those results is therefore very optimistic. On the other hand, when those results are used for cache miss count estimations, the result is very conservative. We can also use the execution frequencies calculated by the IPET analysis for the WCEP instead, which is better suited to estimate the optimization gain. However, in this case the actual cache costs can be higher than estimated, if additional cache costs at call sites that are not on the WCEP and are thus assigned an execution frequency of zero cause the worst-case path to switch.

To get the execution frequencies of methods and instructions for one execution of the optimization target method (the application entry method or the WCA target method), we multiply the local execution frequencies of all call sites along every path starting at the root of the call graph and calculate the sum over all paths when paths merge. For now the analysis assumes that the application is recursion-free. Back edges in the call graph are ignored. The call graph is traversed in topological order, starting at the roots of the call graph.

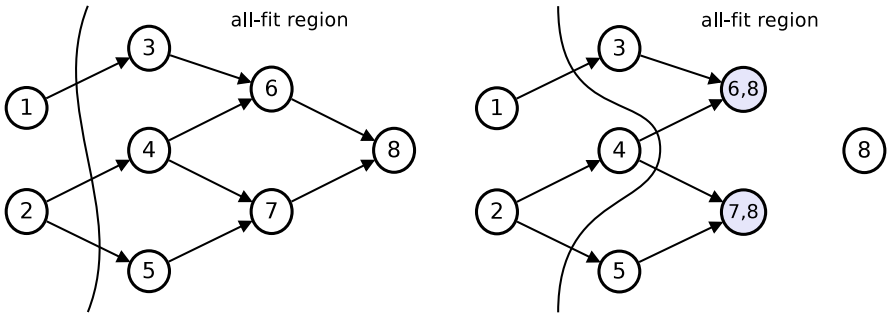
When a node is visited, we sum up the execution frequencies of all call sites of that node to get the execution count for that node. The execution frequencies of instructions in the method of that node is equal to the execution count of that node multiplied by the local execution frequency of the instruction. For virtual calls we over-approximate the execution frequencies by assuming that every execution of a call site invokes every possible implementation.

Method Cache Analysis

To estimate the cache costs, a method cache analysis has been implemented. The method cache analysis provides several analysis modes with varying precision. Besides estimating the total number of cache misses for every cache access, the analysis calculates the total cache miss cost difference when the size of a method is changed.

In this section, we use N for the number of blocks in the method cache, and $\mathit{maxBlocks}(n)$ for the number of cache blocks accessed during the execution of node n. The set of reachable methods $\mathit{reachable}(n)$ represents all distinct methods that can be reached from node n in the call graph, including the method represented by n itself. This analysis uses the sum of cache blocks required by all methods in $\mathit{reachable}(n)$ to over-approximate $\mathit{maxBlocks}(n)$.

For brevity, we call the subgraph of the call graph that consists of all nodes



(a) Initial call graph

(b) Call graph after node 8 has been inlined into nodes 6 and 7

Figure 4.5: All-fit region of a call graph

n with $maxBlocks(n) \leq N$ the all-fit region of the call graph. Figure 4.5a shows an example of an all-fit region in a call graph where every node requires exactly one cache block, and the cache consists of four cache blocks. The all-fit region in this example consists of the nodes 3 to 8. Figure 4.5b shows the impact of inlining on the all-fit region. If we inline all call sites of node 8 in this example, nodes 6 and 7 require an additional block. Node 4 is no longer in the all-fit region, since the code of node 8 now appears two times in the reachable methods of node 4.

Note that in our case context-sensitivity of the call graph is expressed via different nodes for different call strings, not by labeled edges. This means that if a path exists between two nodes, the node at the end of the path is always assumed to be executed by the node at the beginning of the path, the edges themselves are not context-sensitive. Therefore $maxBlocks(n)$ never increases along any edge or path in the call graph, since for every edge $v \rightarrow u$ we have $reachable(u) \subseteq reachable(v)$. The nodes of loops in the call graph must therefore be either all inside or all outside the all-fit region, since $maxBlocks(n)$ must be the same for all nodes n of a loop, or else we would have a path in the loop from a node m to a node m' where $maxBlocks(m) < maxBlocks(m')$.

The cache analysis currently supports the following analysis methods:

- **Always miss**: Every cache access is classified as *always miss*. This is a very pessimistic analysis and provides a simple upper bound on the cache costs. The execution frequency analysis presented above is used to estimate the number of cache misses. The advantage of that analysis is that it is very fast, and changing the size of a method only affects cache costs of cache accesses in adjacent call graph nodes.

- **Always hit**: Every cache access is classified as *always hit*. This analysis never returns any cache miss costs, the optimizations therefore ignore the impact of the method cache on the optimization gain.

- **Always miss or at most one miss**: Let n be the node that contains the cache access. If n is within the all-fit region, the access is classified as *at most one miss*. All other cache accesses are classified as *always miss*. The number of cache misses is estimated using the execution frequency analysis. For *always miss* cache accesses, the number of cache misses is again equal to the execution frequency of the cache access. For the other cache misses, we need to calculate the number of times the application can enter the scope for which the cache access is classified as *at most one miss*. This is done by calculating the sum of the execution frequencies of all call sites classified as *always miss* that invoke a method in the all-fit region that can reach n in the call graph. For LRU caches, the analysis can classify *at most one miss* cache misses as locally persistent. This analysis is similar to the cache analysis performed by the WCA.

  This analysis still provides an upper bound on the cache costs, but it is more precise than the *always miss* analysis. The disadvantage of this analysis is that calculating the number of cache misses is more costly, and even a small code size increase can cause large total cache cost increases, if some methods cannot be classified as *at most one miss* anymore. If the all-fit region changes, the number of cache misses can change throughout the whole all-fit region.

- **Always miss or always hit**: This analysis is similar to the previous analysis method, except that cache accesses within the all-fit region are classified as *always hit* instead of as *at most one miss*. As a result, the analysis no longer calculates an upper bound on the cache miss costs, but the cache miss cost estimation is faster since we do not need to calculate the number of entries into the all-fit region, and the cache miss cost increase is lower when the all-fit region changes. Another advantage is that if the all-fit region changes, the upper bound on cache misses of cache accesses in the all-fit region does not change, therefore we do not need to recalculate all gain estimations in the changed all-fit region.

For the *always hit* and *always miss* analysis, no initialization is needed. For the other analysis modes, the analysis calculates *reachable*(n) and *maxBlocks*(n) during initialization. This is done by first constructing the transitive closure of the call graph. The transitive closure of a graph contains the original graph, as well as additional edges between any two nodes that are connected by a path in the original graph. The transitive closure can be constructed by iterating $\lfloor \log_2 |V| \rfloor + 1$ times over the graph and adding edges between any two nodes if a path of length 2 exists between the nodes. The implementation of that algorithm requires that the original graph contains no self loops, i.e., the original graph must be a simple graph. However since we always include n in *reachable*(n) by definition anyway, we can simply remove self loops before constructing the transitive closure. The sets *reachable*(n) can now be constructed

for every node n simply by collecting all direct successors of n in the transitive closure, the value of *maxBlocks*(n) can be over-approximated by calculating the sum of cache blocks required by the methods in *reachable*(n).

Estimating Cache Cost Increases

The cache analysis is also used to estimate the impact of an optimization on the cache costs. The analysis provides a generic method to calculate the cache miss cost changes due to code size changes. It takes the modified method, the estimated code size difference, as well as methods that will no longer be invoked by the method after the optimization as arguments, and calculates the total method cache miss cost changes, using the execution frequency analysis to estimate the number of cache misses for cache accesses classified as *always miss* or *at most one miss*. This cache analysis only considers the cache costs directly related to the code size increase of a single method. Other cache costs specific to inlining, such as the increase of return costs of call sites in the callee, are calculated by the inliner. To simplify the implementation, only code size increases are currently handled.

The cache cost increase depends on the analysis mode. If the analysis classifies all cache accesses as *always hit*, the result is always zero. If all cache accesses are classified as *always miss*, or if the modified method is not in the all-fit region, the invoke cache miss costs of all call sites of the method and the return cache miss costs of all call sites in that method change proportionally to the code size difference[1] multiplied by the sum of the execution frequencies of the cache accesses.

If the modified method is within the all-fit region, and if the analysis classifies all accesses in the all-fit region as *always hit*, the cache costs only change if the all-fit region changes. The analysis needs to find out how the values of *reachable*(n) and *maxBlocks*(n) are affected by the optimization for all nodes in the call graph. The modified method m only appears in *reachable*(n) if there is a path from node n to a node of m. Therefore we only need to consider nodes that are reachable from m in the reversed call graph.

If m requires i more blocks after the modification, we need to add i to *maxBlocks*(n) for every node n that can be reached from m in the reversed call graph. If a method a is no longer invoked by m, the analysis must find all nodes n where a is no longer in *reachable*(n) after the code modification, i.e., all nodes n that can reach m but can no longer reach a.

If a method is removed from any set *reachable*(n), *maxBlocks*(n) must be updated accordingly. All nodes n where *maxBlocks*(n) $\leq N$ not longer holds (N being the number of available cache blocks), are classified as *always miss* instead of *always hit*, the cache costs increase accordingly. Similarly, for all

[1]For virtual invokes, the largest method in all receivers of the call site determines the cache costs, so there might not be a cache cost change if larger methods can be invoked at the call sites of the modified method.

nodes where $maxBlocks(n) \leq N$ only holds after the optimization, the cache costs decrease.

If the analysis classifies accesses in the all-fit region as *at most one miss*, we also need to handle the increase of invoke miss costs and return miss costs of the modified method similar to the *always miss* analysis, but with a lower number of cache misses. Furthermore, changing the all-fit region may lead to different cache miss counts for all methods that are reachable from any call graph node for which the all-fit region classification changed. The cache costs change proportionally to the difference in the number of all-fit region entries times the cache miss costs of all methods that are reachable from the reclassified nodes.

WCA Integration

The WCET analysis is used to find the current worst-case execution path. The candidate selector uses this information to choose only candidates that are on the worst-case path and can therefore reduce the WCET when optimized. The current greedy algorithm implementation uses the IPET-based WCET analysis method, which calculates the WCET for every node in the call graph. Existing WCA results are reused to calculate the WCET of a call site.

The analysis currently does not allow loops in the analyzed call graph. We can therefore initialize the analysis by traversing the call graph in reverse topological order and analyze the WCET of nodes as we visit them. The topological order guarantees that WCET results exist for all methods invoked by the visited node. Since usually the WCA is not used to analyze the whole application but only the real-time part of the application, a separate call graph is constructed that starts at the target method of the analysis, i.e., the entry method of the real-time code. This call graph is not necessarily a subgraph of the application call graph if the call graphs are context-sensitive, since the root of the WCA call graph starts with a single node with an empty call string, while the application call graph may contain several nodes with non-zero-length call strings representing the target method in different execution contexts.

Since the optimizer and the WCA use the same data structures for optimization and analysis, exchanging data is trivial. The WCA calculates worst-case execution frequencies and attaches them to the basic blocks of the CFG of the analyzed methods. To check if an instruction is on the WCEP, the optimizer only needs to check if the attached worst-case execution frequency is greater than zero.

Gain Estimation and Rebate Ratios

The candidate selection is based on the rebate ratio of an optimization candidate, which is simply the estimated total gain of an optimization divided by

the code size increase. Therefore we need to calculate the gain and the code size increase of all optimization candidates.

In order to calculate the total estimated gain g of a candidate, the inliner provides the execution time gain g' for a single execution of the optimized call site excluding cache costs (i.e., the invocation overhead), as well as the total cache cost changes c' within the optimized code. The execution count f of the optimized code is provided by the execution frequency analysis. The cache cost increase c in the rest of the application due to the code size increase of the modified method is estimated by the method cache analysis. The total gain g of an optimization candidate is then estimated as

$$g = g' * f - c' - c$$

The application code size increase Δs is equal to the code size increase of the modified method, minus the size of methods that are no longer used after optimization, provided that removing unused code is enabled. Using the estimated gain and code size increase, we can calculate the rebate ratio r of an optimization candidate as $r = g/\Delta s$.

However, the estimated total gain and thus the rebate ratio are over-approximations of the actual WCET gain, if the optimization causes the WCEP to change, as discussed in the example in Section 3.2. It might be possible to estimate the WCET gain for every inlining candidate by reducing the execution time of the call site to inline by the estimated gain g' and to calculate the WCET for that modified control-flow graph. However, performing a WCET analysis for every candidate is very expensive and has not yet been implemented. Furthermore, the WCET gain of the application might still be lower than the WCET gain of the optimized method, since decreasing the WCET of a method may result in a change of the WCEP in the invokers of that method too.

Candidate Selection

Once all candidates have been found and evaluated, the greedy algorithm needs to select one of those candidates for optimization. If the WCET analysis is not used, the algorithm simply selects the candidate with the highest rebate ratio first. In this case the optimizer is similar to an ACET optimizer.

If the WCET analysis is used, the algorithm selects only candidates that are on the worst-case path. By default the algorithm chooses between candidates from all methods that are reachable from the WCA target method and that are on the worst-case path in the call graph. If the analysis is configured to iterate over the methods in the call graph in top-down or bottom-up order, the analysis only chooses between candidates within a single method at the same time. Optimizing methods can change the global WCEP, but if the call graph is visited in a fixed order, methods that appear on the WCEP after they

have been visited are not revisited. Therefore when methods are optimized in a fixed order, the algorithm only considers the local worst-case paths of the methods, regardless of the global worst-case path.

4.5 Updating the Call Graph and the Analyses

After a call site has been inlined, the call graph needs to be updated. If the call graph does not contain call site references, this is fairly simple. New edges need to be added from the invoker to all children of the callee. The edge from the invoker to the callee needs to be removed if the optimized call site was the last call site of the callee in the invoker.

If the call graph is context-sensitive, new nodes need to be created since the call strings consist of call site references (see Section 2.4). All nodes in the call graph with a call string containing the removed invoke site need to be removed. Instead of adding edges from the invoker to all direct successors of the callee, new nodes need to be created for the inlined call sites.

This can be done in the following way: We iterate over all nodes $\langle c, m\rangle$ representing a context of the optimized method m with call string $c = \langle c_1, \cdots, c_n\rangle$. For every direct successor $\langle c', m'\rangle$ of the inlined callee we construct a new call string $c'' = \langle c_1, \cdots, c_n, c_m\rangle$ where c_m is the new call site in the inlined code corresponding to the call site in the callee that invokes $\langle c', m'\rangle$, thus removing the inlined call site and replacing the call site in the callee with the call site in the inlined code in the call string c'. If the length of c'' is longer than the maximum call string length k, we remove the first entry from the call string.

An edge from $\langle c, m\rangle$ to $\langle c'', m'\rangle$ is created. If the node at the head of the new edge does not exist, it is created first. For every direct successor of $\langle c', m'\rangle$ a new successor of $\langle c'', m'\rangle$ is added. The call string of the new children are constructed by appending the top entry of the call string of the old child to c'', thus replacing the prefix containing the original call site with the new prefix containing the corresponding inlined call site. This is performed recursively on all new nodes. All nodes reachable from the callee over any path of length up to k are copied in this way. All other nodes do not contain call sites of the callee or the optimized method.

The execution frequency analysis, the method cache analysis, and the worst-case analysis need to be updated for the optimized method as well as any new nodes in the call graph.

For a context-insensitive call graph, only the execution count of the invoked method changes. Since we removed a call site of that method, the execution count of the method is decreased by the execution count of the removed call site. The execution counts of the inlined instructions is equal to the execution count of the removed call site times the execution frequency of the corresponding instruction in the callee (relative to one call of the callee). The execution count of the instructions in the callee decreases by the same

amount. Therefore the execution counts of the children of the callee do not change, since the sum of the execution counts of the call sites in the callee and in the inlined code is the same as the execution count of the call sites in the callee prior to optimization.

If the call graph is context-sensitive, we additionally need to calculate execution counts for all new nodes. This is done the same way as in the initialization of the execution frequency analysis, except that the call graph is traversed by starting at the optimized method instead of at the root, and only nodes for which the distance to the optimized method is at most k (the maximum call string length) are visited. As in the initialization, we ignore back edges. The nodes are visited in topological order, the sum of the execution counts of all call sites of a node is used as the execution count for new nodes.

Updating the cache analysis requires analyzing the new nodes and removing the callee from the set of reachable implementations of all nodes that no longer have a path to the callee, if no other call site in the optimized method invokes the inlined method.

To calculate the sets of reachable methods for the new nodes, first a subgraph of the call graph is created, which contains all paths of length $k + 1$ starting at the nodes of the optimized method. The subgraph therefore contains all new nodes, and every path either ends in a new node with no children or in a node for which the set of reachable methods is known. Similar to initialization, the sets of reachable methods are generated by creating the transitive closure of the graph and then adding all children of a node in the transitive closure to the set of reachable methods, with the exception that for all leafs the set of methods reachable from the leaf is added too.

If the callee is no longer invoked from the modified method, the callee is removed from the sets of reachable methods in the same way as the analysis checks for reachable set changes to find all-fit region changes after code modifications. Note that we do not need to increase any set of reachable methods after inlining, because all methods reachable from the callee are also reachable from the invoker. Therefore copying call sites from the callee to the invoker does not add new methods to the set of known reachable methods of the invoker.

Finally, the sum of cache blocks required for all methods reachable from a node needs to be increased by the number of additional cache blocks required by the optimized method. This is done for all nodes that have a path to the optimized method in the call graph. This can change the cache miss classifications and thus the estimated cache miss costs of call sites anywhere in the call graph, depending on the used analysis. The gain estimations need to be updated accordingly.

To update the WCA results, a subgraph of the edge-reversed call graph is created first. This subgraph contains only methods reachable by the WCA

target method[2] and only nodes reachable (in the edge-reversed graph) from any of the new nodes, the nodes of the optimized method, or methods for which the results of the cache analysis changed, i.e., it contains all methods that may call methods whose WCET bound may have changed. The WCA currently does not allow recursion, therefore we can assume that the resulting graph is acyclic. Similar to the initialization of the WCA analysis, the graph is then traversed in topological order, and for every visited node a new WCET and WCEP is calculated.

4.6 Other Approaches to the Call Site Selection

The greedy algorithm is a simple method to select call sites for inlining. It has several shortcomings though. It does not calculate the total gain of sequences of optimizations. For instance, the algorithm may choose not to inline any other call site, because doing so would cause cache costs larger than the invoke overhead due to some methods not being classified as all-fit anymore. It ignores the possibility that doing so could create new optimization candidates which it might be able to inline with only very low cache costs if the optimized method does not require additional cache blocks, and could thus lead to a better total gain.

As mentioned before, the implemented greedy algorithm does not use the actual WCET gain to select its candidates and may actually increase the WCET due to cache costs. To avoid this behavior, deoptimization could be used to restore the original call site if the WCET analysis shows that the WCET increased.

A different approach for call site selection is presented in [11]. The inliner decides on which call sites to inline based on a variety of features such as the caller and callee size, the WCET of the caller and the callee, or the number of live registers at the call site, using decision trees. The decision trees are generated by using a set of example applications. The WCET gain of inlining a call site is determined by analyzing the WCET of the unoptimized version and a version where only the call site to analyze has been inlined. The results are passed to the learning algorithm that creates a set of decision trees. During optimization of a program, the decision trees classify the call sites of the program, the final inlining decision is done by a majority vote on the decision tree results. The advantage of this method is that the optimization heuristic can be trained for different target architectures automatically. The compiler designer does not need to find good heuristic parameters himself.

The IPET based WCET analysis [28] could also be used to classify call sites directly. The WCA creates separate paths for a call site in the control-flow

[2]This is done by keeping a separate call graph that starts at the WCA target method and that is used for all WCA related call graph searches. This call graph is also updated during inlining.

graph, representing the costs of an invocation with and without cache miss costs. Constraints are used to limit the number of cache misses depending on the cache analysis.

Inlining a method can be represented in the control-flow graph by adding a third path to the call site that skips the invoke instruction and has the execution time of the generated prologue attached to it. Constraints need to restrict the solution to take either the optimized path or the unoptimized path for all executions, depending on the state of a decision variable. The cache costs at the call sites of the optimized method and at the return edges into the optimized method now depend on the value of the decision variables.

The WCET of a control-flow graph with N basic blocks can be expressed as

$$\text{WCET} = \max_{\langle e_1,\cdots,e_N\rangle \in \mathbb{N}_0^N} \sum_{i=1}^{N} c_i e_i$$

where c_i is the cost of the basic block B_i and e_i denotes the execution frequency of that block. The values of the execution frequencies are bounded by loop bounds and constraints encoding the fact that the sum of the execution frequencies of the ingoing edges of a block must be equal to the sum of the execution frequencies of its outgoing edges.

The optimizer needs to minimize that expression using the decision values, therefore we need to solve

$$\text{WCET} = \min_{\langle d_1,\cdots,d_k\rangle \in \mathbb{Z}_1^k} \max_{\langle e_1,\cdots,e_N\rangle \in \mathbb{N}_0^N} \sum_{i=1}^{N} c_i e_i$$

for a method with k call sites. The problem with this approach is that this can no longer be solved by linear programming. Furthermore, not only are the cache costs expressions in the decision variables d_i instead of constants, the cache analysis results and thus the constraints on the number of cache misses at a call site also depend on the decision variables.

Another problem is that this approach does not cover the fact that inlining a method can create new call sites. It might be possible to encode this in the supergraph[3] by adding additional decision variables to not only decide if a call site is inlined but also which of the methods reachable in the call graph should be recursively inlined at that call site. This however results in a large increase in the number of decision variables and in the size of the constraints and cache cost expressions. So this approach might not be practical except for small programs or for the classification of call sites for only one method at a time. It should be noted though that this method could also be used for

[3]A supergraph is a graph that contains all control-flow graphs where basic blocks are split at call sites and additional edges from the call site to the entry block and from the exit block back to the call site are added for the control-flow graphs of all implementing methods of a call site.

optimizations other than inlining by adding different paths to the control-flow graph representing optimized and unoptimized paths. The feasibility of this approach however has not been studied in this thesis.

Using the IPET based analysis for classification of optimization candidates makes the border between WCET analysis and optimizer less clear, which can be regarded as a disadvantage since this is in conflict with the goal of creating separate, specialized tools for a single task like analysis or optimization.

The most important difference between this approach and the inliner presented in this thesis or the inliner presented in [11] is that instead of using the WCET analysis to provide feedback to the optimizer, the optimizer provides a set of optimization candidates to the WCET analysis. The WCET analysis then classifies the candidates itself.

The advantage of this is that the effects of an optimization on the WCEP need to be handled only by the WCA, which can be done more efficiently than by a classification algorithm that needs to switch back and forth between the optimizations and the WCET analysis. The cache and timing analyses do not need to be exposed by the WCA.

An interesting use case is function splitting. Since the method cache has a bad performance if a large method is called where a large part of the code is not on the WCEP, extracting that code into a separate method and replacing it with a call could reduce the cache costs on the WCEP. However, if the WCET of the removed code is only slightly below the execution time of the WCEP, adding a call site might actually increase the overall WCET. Similarly, inlining a method increases the return cache miss costs of call sites in the optimized methods. If those call sites do not lie on the WCEP, the additional costs can be ignored, but only up to the point where the WCEP switches to those call sites. To handle this, the optimizer would either need to deoptimize the code if the WCA showed that the WCET has actually been increased, optimize very conservatively, or it would require timing information on paths other than the WCEP too. By solving the classification using the IPET based approach, this is handled exclusively by the algorithm to solve the min-max problem.

On the other hand, either the optimizer needs to describe its code transformations to the WCA so that the WCA can calculate the effects of an optimization on the cache and the WCET and to update the flow facts, or the WCA needs to know how the optimizations affects the code itself, which makes the implementation of the analysis more complex. An alternative could be to implement a separate, possibly less precise cache- and WCET analysis (for driving the optimizations we do not need the final precise WCET, it does not even need to be safe as it is only used to drive the optimization heuristics), designed for optimization candidate classification. The more precise WCET analysis is used only to analyze the WCET of the optimized application, or to create an initial solution for the input of the optimizer that could then be used by the second WCET analysis.

Chapter 5

The Framework

During the work on this thesis a framework for all analyses and tools was created to implement the optimizations, to simplify the development of new tools, and in order to let all analyses and optimizations work on the same data structures in memory. The existing worst-case analysis and the data-flow analysis have been ported to the new framework so that the optimizer can run the analyses iteratively on the modified bytecode without the need to export the transformed code prior to the analysis. Existing results that are not affected by the optimizations can be reused by the analyses without the need to keep results and data structures like the call graph consistent over separate tools.

This chapter gives an overview over the features of the framework and presents the toolchain used to create optimized code for JOP and to analyze the worst-case execution time of the code.

5.1 Framework Overview

The framework is implemented in Java and uses BCEL[1] for reading and writing class files. It provides implementations to find and load all classes required by the target program (see Section 5.3), construct and modify call graphs and control flow graphs as described in the previous chapters, and to simplify lookups and modifications of constant pool entries and class members (i.e. methods, fields and attributes).

Additionally the framework provides code for common tasks such as:

- **Configuration, logging, and setup of tools**: This includes loading and parsing of configuration files and command line arguments. Common configuration options are provided and handled by the framework,

[1] The Apache Commons Byte Code Engineering Library: `http://commons.apache.org/bcel/`

so that all tools in the tool chain have similar configuration options and can use the same configuration files. However, a user provided configuration is not loaded by default, any configuration to use must be specified at the command line. Therefore default values are not changed unknowingly by any leftover configuration files, which ensures reproducible results.

- **Lookups in the class hierarchy and access checks**: Methods are provided to resolve fully qualified class names, field names and method signatures to a reference to the correct member object. The framework supports common tasks such as finding overriding methods or super methods.

  Access checks are required to determine if a method actually overrides a method defined in a super class or if a call site can be copied to another class by the inliner without violating the access restrictions.

- **Finding possible implementations for call sites**: The set of methods that can be invoked at virtual invokes is determined by a lookup in the class hierarchy or the call graph if available. The result can be made more precise by thinning the call graph first, e.g., by using receiver type information from the data-flow analysis to construct the call graph.

  The receiver of non-virtual invocations (`invokestatic` and `invokespecial` as well as bytecode instructions that are implemented as Java methods in the JVM) can be determined statically. Super method invokes are resolved as specified by the `invokespecial` instruction (see Section 2.2). Dynamic invokes as introduced by Java 7 are not handled.

- **Common graph algorithms**: Various simple traversal and graph transformation algorithms are provided, e.g., to traverse the classes in the class hierarchy, to visit all elements of a class, to traverse the call graph in depth-first order, or to detect back edges in the call graph, or to construct acyclic graphs from directed graphs by removing back edges.

- **Nested classes support**: To determine the enclosing classes of nested classes the `InnerClasses` and `EnclosingMethod` class attributes need to be used. Since the `InnerClasses` attribute of a class needs to contain an entry for every nested class referenced by the constant pool of the class, it may be necessary to add new entries if new constant pool class references are added, e.g., due to inlining of a method from another class.

  The access checks also require information about nested classes, since for instance private methods of nested classes can be accessed by their enclosing classes whereas members of a local class (i.e., a class that is defined within a method) can only be accessed in the enclosing method.

- **Processor models**: Basic information about the target JVM and processor are provided, such as the names of classes that are required by the JVM, and must not be removed, the maximum method size or the maximum number of local variables and references to methods that are invoked to execute certain Java bytecode instructions.

- **Source code line number handling**: The frameworks provides methods to keep accurate source line numbers and source file references for instructions even after inlining of methods from different classes, see also Section 5.5.

- **Java 6 support**: In order to remove unused constant pool entries all used entries must be found. This means that all elements of the class file that should be kept must be parsed completely, and all other elements must be removed. BCEL was therefore extended to support all standard Java 6 attributes like `EnclosingMethod` or various annotation attributes.

Timing models are not included in the framework. Those are implemented exclusively by the worst-case analysis tool. Therefore only the WCA tool needs to have a detailed timing model of the target processor. The optimizer uses the WCA tool to get timing information for code sequences and details about the cache if required.

Code Representation

In contrast to other optimization frameworks like Soot [33], bytecode is used as the only internal representation. Optimizations work directly on bytecode, there is no simplified stack instruction set or a register architecture representation like quadruple code (BCEL makes a few minor simplifications, e.g., instructions like `aload_0` are represented by the parametrized version like `aload_n`, but every bytecode instruction in the class file is represented by exactly one BCEL instruction and the opcode and size of the bytecode instruction that will be generated can be determined unambiguously by looking at the parameters of the instruction). The framework must not modify the code on its own during class loading or writing and must present the instruction sequences the same way to the analyses and tools as they are stored in the class files. Otherwise the worst-case analysis results would become inaccurate or even unsafe.

By using the Java instruction set architecture as internal representation, the tools always work on code that is very close to the code that will be executed on the processor (invokes of native JVM methods are replaced by custom bytecodes at the end of the JOP toolchain). However, since standard Java bytecode instructions are used, the tools are not limited to a specific Java processor by the instruction set. Since on JOP the execution time of instructions is independent of previously executed instructions (except for invoke and

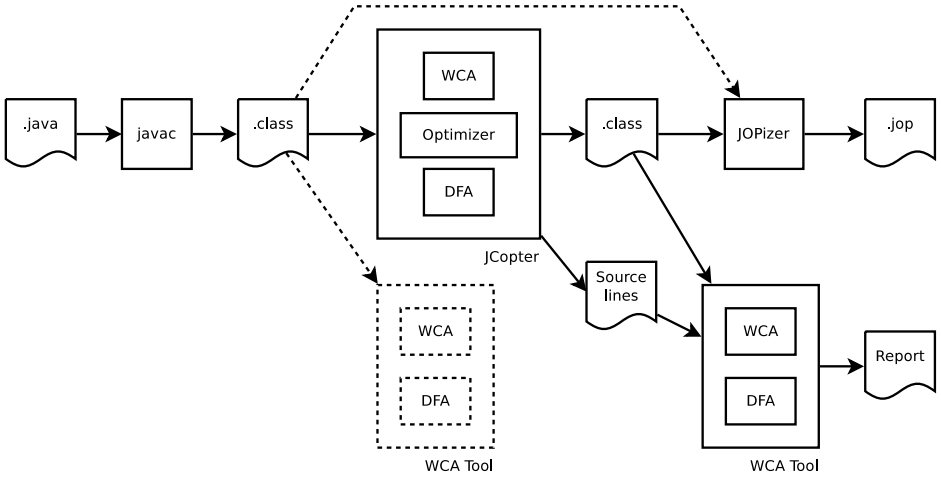


Figure 5.1: JOP Toolchain

return instructions whose execution time also depends on the method cache contents that depends on the sequence of executed invokes) the execution time of bytecode instruction sequences are composable. This allows the worst-case analysis to calculate accurate timing information for any given sequence of instructions, independent of the surrounding instructions.

5.2 Tools

The JOP toolchain is shown in Figure 5.1. The standard Java compiler `javac` of the Java Development Kit (JDK) is used to compile Java sources to Java class files containing bytecode, but other compilers emitting JVM class files could be used as well.

The JOPizer tool [24] takes either optimized or unoptimized files (if the optimizer is not used) as input and generates a .jop file containing the bytecode, the constant pools and information about the structure of classes, methods and fields that can then be uploaded to the JOP processor.

The worst-case analysis tool takes the Java class files as input and uses the source code line numbers from the class files (and additionally from the previously stored source line database file for optimized code) to read source code loop annotations. The loop annotations are merged with the loop bound results produced by the data-flow analysis. The WCA tool then performs a worst-case execution time analysis for a given method and generates a report containing the calculated worst-case execution time and the source code annotated with the worst-case path.

JCopter

The worst-case oriented bytecode optimizer JCopter takes class files as input, invokes the data-flow analysis and uses the worst-case analysis to guide the optimizations and emits new Java class files containing the optimized code, without unused methods or classes. Source line references that cannot be stored in class files directly (i.e., line numbers for code with a different source file than the source file of the class containing the code) are stored in a separate file.

The optimizer currently executes the following steps:

1. Optionally perform a receiver type data-flow analysis
2. Invoke SimpleInliner to eliminate getter, setter and wrapper methods
3. Rerun the data-flow analysis to analyze receiver types and loop bounds, if enabled
4. Rebuild the call graph using the data-flow results if available
5. Initialize the WCET analysis, initialize the execution frequency analysis and method cache analysis
6. Perform inlining using the WCA-driven greedy selector
7. Remove unused class members and classes
8. Remove unused constant pool entries
9. Write the application code and the source line references to files

Although the WCA instance in the optimizer could also be used to generate a WCET report on the optimized code, running the WCA as a separate tool makes it easier to use a different (i.e. more precise) WCET analysis method than the optimizer does, and the correctness of the worst-case analysis does not depend on the correctness of the implementation of transformations of data structures like the call graph or the type graph in the optimizer. Incorrect line number references however could make the WCA results unsafe.

5.3 Class Loading

In order to build a type graph and a call graph, all classes that are used by the application are loaded by the framework before any analyses or code optimizations are performed. This is done by following all class references in the constant pools of the class files recursively, starting at the application main class as well as the JVM implementation classes (the classes that contain the boot method and the methods that are invoked to execute bytecodes that are implemented as Java methods). Class names in parameter types of methods need to be checked too, since the parameter types of methods are stored as

descriptor strings only and not as class references. So the classes of parameters do not occur as class reference in the constant pool if those types are not elsewhere in the class too.

The class loader will find all classes used by the application and the JVM, since every class that is instantiated or referenced somewhere in a used method or field needs to have a class reference entry in the constant pool associated with the method or field, assuming that reflection is not used to instantiate or access a class. This assumption can be enforced by checking that no class or method related to reflection is used by the application code.

The loader does not check if methods of the classes to load are actually used. Classes that are referenced by unused methods or dead code are loaded too. Unused classes and methods are removed in a separate step as described in the following section.

5.4 Removing Unused Elements

To reduce the total size of the application and to remove class members and constant pool entries left over by optimizations, two cleanup algorithms have been implemented, one to remove unused classes, methods and fields and another one to remove unused constant pool entries. They are executed by the optimizer before the class files are written. Unused (debug) attributes are removed first so that as many constant pool entries as possible can be removed.

Unused Members Remover

Classes can contain class members (methods and fields) that are never used by the application, either because the application code (including libraries, the Java runtime library and the JVM implementation itself) contains unused methods or fields or because the optimizer inlined a method at all of its call sites. Classes that are only referenced by unused methods can be removed too.

To remove unused members, all used members are marked first, starting at all entry methods like the JVM startup method, the application main method and `run()` methods of thread classes. All methods that may be invoked by a method as well as all fields that are accessed by a method are marked too, and searched recursively. All classes that are referenced in a method descriptor, a field type, as superclass or implemented interface as well as classes containing marked methods are marked as used too. If a class is marked and contains a static initializer method `<clinit>()`, the static initializer is processed too.

Finally, all unmarked classes and class members can be removed, with a few exceptions:

- Fields of hardware objects (classes that are used to map IO device regis-

ters to object fields) [26] must not be removed, else the mapping between the fields and the register addresses is destroyed.

- Fields with constant integer values are not removed because they may be used by source loop bound annotations.
- Non-abstract methods that are only referenced by invoke instructions but are never invoked are kept but their code attribute is removed and the method and its class are made abstract.

  This can happen if the declared type of a virtual invoke instruction provides a method implementation for the method reference, but the data-flow analysis proved that all possible receivers of the call site override the method in the declared type. To handle this case, the algorithm marks methods as `referenced` when they appear as reference at a call site and are not marked yet. Methods are only processed when they are marked as `used` the first time (i.e., when they appear as implementing method in the call graph at a call site).

Constant Pool Cleanup

If class members or instructions are removed, entries in the constant pool can become unused. Since the underlying BCEL library does not provide the means to modify existing entries in the constant pool (this would in fact require the user to check if the entry he wants to change is not used anywhere else where the old value is still used), unreferenced entries can also appear due to code transformations.

To remove such entries, a simple algorithm can be used: First all used entries are marked by visiting all elements of the class structure (i.e. the class, its members and their various attributes like the code, the InnerClasses attribute,...), then the constant pool is rebuilt using only the marked entries. If there is at least one unused entry in the old pool prior to a used entry, this will cause at least some indices to change. Hence we need to update all constant pool references in the class to the new indices too.

Although it would be possible to do this in a single pass by adding entries to a new constant pool and updating the references to the constant pool whenever a reference is encountered during a traversal of the class structure, there is a possibility that updating the constant pool references can increase the code size if the new constant pool has more than 255 entries. If an instruction (e.g. `ldc`) refers to an item with an index of less than 256, then the instruction can be encoded with two bytes. If the entry gets a new index larger than 255, then the same instruction requires four bytes (due to the `wide` opcode). This can increase the cache miss penalty of the method cache and therefore increase the WCET of the application, and if the new method code size becomes larger than the maximum code size, the application cannot be executed on the target anymore.

Therefore the entries are added to the new constant pool in the same order as they appear in the old constant pool after all used entries are found and before the references are updated. Since the new constant pool only contains a subset of the entries of the old constant pool, no entry in the new constant pool has a larger index than its corresponding entry in the old constant pool, and so the method code size is not increased.

5.5 Source Line Numbers

To calculate a static WCET, the worst-case analysis requires loop bounds for all loops. The programmer can annotate loops in the source code to improve loop bounds found by the data-flow analysis or to provide loop bounds if the data-flow analysis is not used or cannot detect loop bounds. To find the annotations, the instructions of the loop header must have a reference to the source code file and the first line number of the corresponding loop statement in the source code. The Java JVM specification defines attributes to store the name of the source file for a class and to map instructions to line numbers, but references to a source file different from the class source file as required after inlining cannot be stored. This can be solved either by defining custom attributes or by storing the information in a database file separate from the class file. To avoid additional entries in the constant pool for attributes that are only used in the toolchain, the latter approach was implemented.

If the optimizer performs loop transformations, the loop bounds need to be transformed too [6]. The new loop bounds need to be stored in a similar way so that the WCA can use the transformed loop bounds instead of the outdated source annotations.

The source line references are also useful to show the programmer which statements in the source code contribute to the WCET in the unoptimized or the optimized code.

Chapter 6

Evaluation

The performance of the implemented WCET-driven inliner was evaluated by analyzing and executing various benchmark applications. The benchmark applications were optimized by the optimizer implemented in this thesis. The WCA was used to calculate a WCET bound for the optimized and the unoptimized applications. The benchmarks are discussed in Section 6.1. Section 6.2 presents the results for various optimization configurations.

6.1 Used Benchmarks

The JemBench suite [27] provides three real-world application benchmarks. Kfl is a controller application used in a rail cargo project. It uses only static methods and no object allocation. The Lift application is a lift controller, which uses some virtual invocations. However, objects are only instantiated during initialization. The UdpIp benchmark is based on an UDP/IP stack that also uses virtual invocations.

In addition to the JemBench application benchmarks, a small software that controls a line-follower robot was evaluated. WCET analysis results for the JemBench application benchmarks and the line-follower software have been published in [28].

To test the optimizer with a larger application, the Java port of the debie1 (First Standard Space Debris Monitoring Instrument, European Space Agency[1]) benchmark was used. The original benchmark is written in C and is based on the on-board software of the DEBIE space debris impact monitoring instrument for satellites. The benchmark has been ported to Java using an object oriented approach. WCET analysis results of the unoptimized benchmark were published in [34].

The debie benchmark consists of several interrupt handlers and periodic tasks. Three interrupt handlers are used for telecommand reception, telemetry

[1] `https://gate.etamax.de/edid/publicaccess/debie1.php`

Name	Description	Methods
LineFollower	Simple line-following robot	9
Kfl	The *Kippfahrleitung* controller	46
Lift	A lift controller	19
UdpIp	A UDP/IP benchmark	39
Debie 1	jDebie telecommand interrupt handler	36
Debie 2	jDebie transmission interrupt handler	24
Debie 3	jDebie hit detection interrupt handler	28
Debie 4	jDebie telecommand execution task	68
Debie 5	jDebie acquisition task	82

Table 6.1: Overview over the used benchmarks

transmission and hit trigger handling. Two periodic tasks process the data provided by the interrupt handlers. The benchmark also includes a monitoring task, but meaningful flow constraints are not available for all loops, therefore this task has not been analyzed.

Table 6.1 gives an overview over the used benchmarks as well as the tasks and interrupt handlers of the debie benchmark. The table also lists the number of methods in the call graph of the real-time code to optimize. This includes methods in the JVM that are called to execute bytecode instructions that are implemented as Java methods.

The WCA tool is used to find a WCET bound for the real-time tasks of the optimized and the unoptimized benchmark applications for the JOP platform, by using the global IPET-based analysis and the *at most one miss* cache approximation. The data-flow analysis is used to find receiver types and loop bounds. Only context-insensitive analyses have been used, since for the simple benchmarks there is no advantage in using context-sensitive analyses, and for the jDebie benchmark a context-sensitive analysis requires too much memory. The benchmarks where executed on JOP to measure the execution time of the tasks. The results give the maximum execution time that has been measured. However, since the tests are not exhaustive and the execution time depends on the cache content of the processor, the measured execution times cannot be guaranteed tp show the real WCET. All analyzed and measured execution time results in this chapter are given in numbers of cycles.

6.2 Results

First all benchmarks were optimized using only the SimpleInliner. Table 6.2 shows the measured execution times and the analyzed WCET of the unoptimized and the optimized benchmarks. The SimpleInliner optimizes every

Benchmark	Unoptimized		SimpleInliner		
	Measured	WCET	Measured	WCET	Inlined
LineFollower	2232	2287	2232 (-0%)	2287 (-0%)	81
Kfl	10040	22251	10040 (-0%)	22251 (-0%)	94
Lift	5007	7603	5007 (-0%)	7603 (-0%)	81
UdpIp	7959	127501	7934 (-0%)	127069 (-0%)	97
Debie 1	6977	17717	6043 (-13%)	15518 (-12%)	564
Debie 2	6601	9104	6177 (-6%)	8409 (-8%)	564
Debie 3	67666	132353	61160 (-10%)	123785 (-6%)	564
Debie 4	24652	26863	21496 (-13%)	23063 (-14%)	564
Debie 5	1288962	1382300	659143 (-49%)	750158 (-46%)	564

Table 6.2: Execution times for unoptimized code and after SimpleInliner

method in the code, including unused methods, methods in the JVM, and initialization code. The number of inlined call sites is therefore quite high, even if the actual real-time tasks contain only a few methods. The JemBench applications do not use getter, setter and similar methods, the SimpleInliner therefore has nearly no impact on the execution times for those benchmarks. For the jDebie benchmark, which has been written in a more object-oriented style, the SimpleInliner is able to reduce both the measured execution time as well as the WCET bound by about 6% to 14% for the interrupt handlers and the telecommand task. The execution time of the aquisition task (Debie 5) is nearly cut in half by inlining all small methods.

Optimizing multiple WCA targets at the same time is currently not supported by the optimizer. We therefore start the optimizer for each of the jDebie real-time tasks separately, using the output of the previous optimization run as input of the next run. Optimizing for a different target method could have a negative influence on the WCET of the already optimized target methods. However, this was not the case with the jDebie benchmark. Optimizing any single task of the jDebie benchmark produced the same execution time for that task as when all tasks of the benchmark were optimized.

Evaluation of the Greedy Inliner

After the initial optimization with the SimpleInliner, the greedy inliner was used to optimize the remaining call sites that were not handled by the SimpleInliner. Figure 6.1 and Figure 6.2 show the effect of the greedy inliner on the execution time of the benchmark applications for various configurations of the inliner. The figures plot the WCET bounds analyzed by the WCA tool, the maximum execution times measured on JOP, and the cache miss costs as estimated by the WCET analysis, relative to the WCET results achieved by

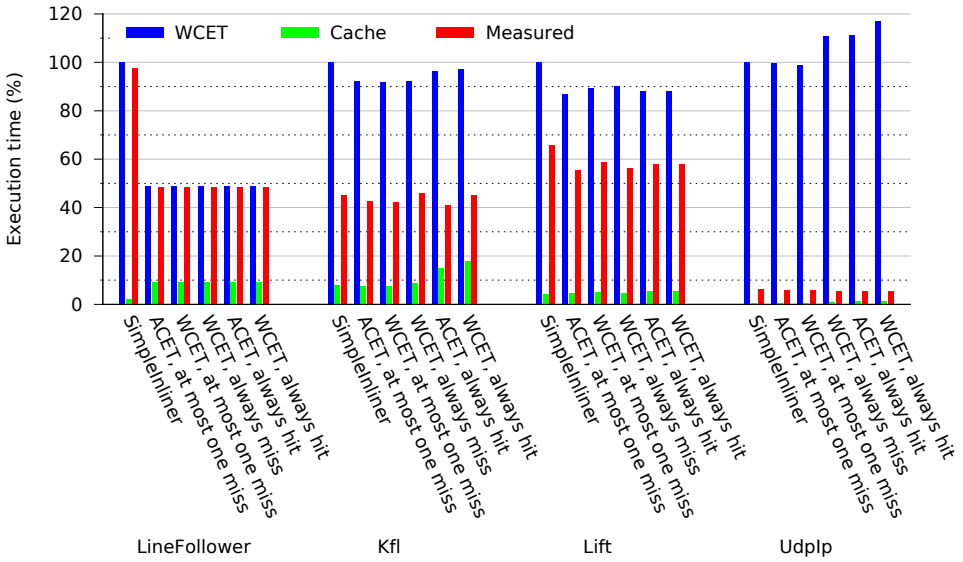


Figure 6.1: Execution times for various greedy inliner settings

applying the SimpleInliner only.

The following optimizer configurations are used in Figure 6.1 and Figure 6.2:

- **SimpleInliner**: This configuration represents the case where the greedy optimizer is not used, and is included in the figures for reference. The execution times shown for this configuration are the same as in Table 6.2.

- **ACET, at most one miss**: The ACET-driven greedy inliner is used, i.e., no WCA results are used for optimization decisions. The cache costs are approximated using a cache analysis which classifies all cache access in the all-fit region as *at most one miss*. All other cache accesses are classified as *always miss*.

- **WCET, at most one miss**: In this configuration, the WCET-driven greedy inliner is used, i.e., only candidates on the current WCEP are optimized. The cache cost estimation is the same as in the previous configuration.

- **WCET, always miss**: Again, the WCET-driven greedy inliner is used, but all cache accesses are classified as *always miss*, which simplifies the cache cost analysis but provides a much more conservative cost estimation. Inlining only changes cache costs at the caller and the adjacent nodes in the call graph.

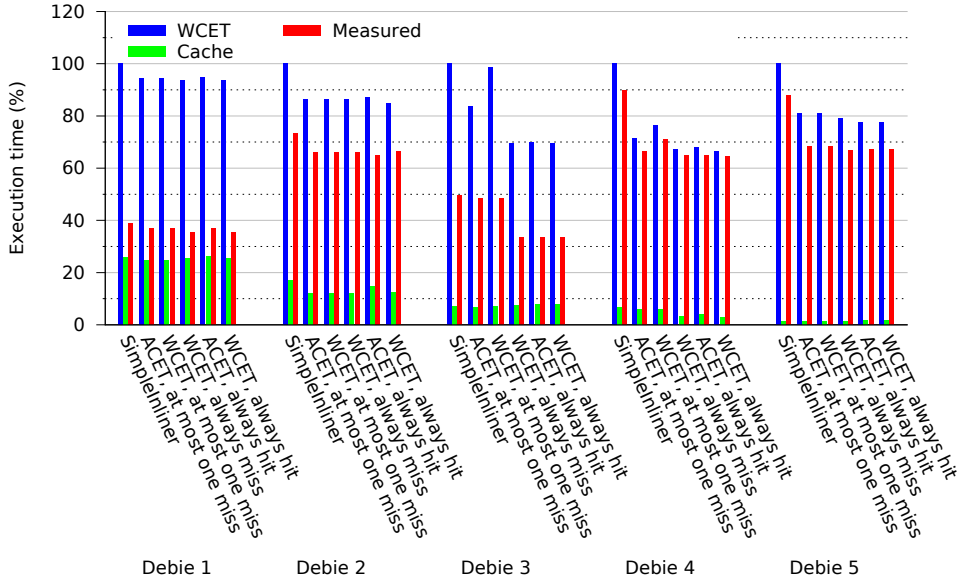


Figure 6.2: Execution times for various greedy inliner settings

- **ACET, always hit**: The ACET-driven greedy inliner is used, and all cache accesses are classified as *always hit*, i.e., the cache costs are not taken into account. This is the most aggressive configuration.

- **WCET, always hit**: Only candidates on the WCEP are optimized, but cache costs are ignored. This is a very aggressive configuration of the WCET-driven inliner, but in contrast to the previous configuration, optimization candidates outside the WCET-critical code are ignored.

In all configurations, we optimize all methods reachable from the WCA target method, i.e., the code that implements the real-time tasks of the benchmarks. The inliner uses a global selection strategy, i.e., it searches for the best candidates in all methods in the real-time code. The call graph is not traversed in a fixed order for optimization.

The LineFollower application is small enough so that the whole mission critical code fits into the cache. The inliner is able to inline almost all candidates. The optimizer configurations have no impact on the set ofi the optimized call sites, the execution times resulting from the optimization are therefore the same for all configurations.

The three benchmark applications of the JemBench suite are large enough so that different inliner configurations can lead to different sets of inlined call sites. However, the benchmarks do not contain many methods that can be inlined without a negative impact on the WCET bound analysis. The more

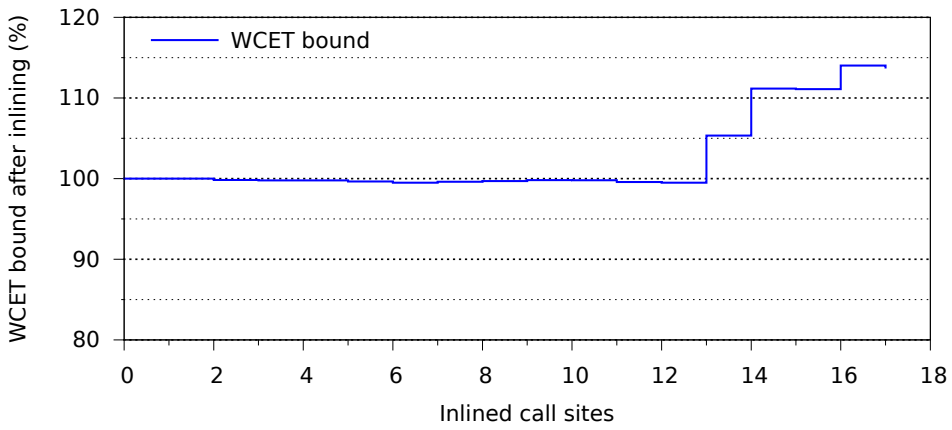


Figure 6.3: Intermediate WCET analysis results during optimization of UdpIp with WCET-driven inliner and *always hit* cache analysis

precise *at most one miss* approximation results in the largest speedups for those applications.

The less precise cache approximations cause the inliner to optimize more call sites, as shown in Table 6.4 below. For the UdpIp benchmark, this even results in a major increase of the WCET bound. Figure 6.3 shows the WCET bounds calculated after every inlining operation by the WCA that drives the optimizer. The final increase of about 15% of the WCET after inlining is caused primarily by only three candidates, for which the optimizer estimated the impact on the WCET incorrectly. If the optimizer undid such individual bad decisions, less precise cache approximations could still be sufficient to optimize the remaining call sites.

For the jDebie benchmarks the situation is quite different. Again the less precise cache approximations lead to more call sites being inlined. However, in contrast to the JemBench applications this has a positive overall impact on the WCET. More aggressive optimization configurations lead to lower WCET bounds as well as lower measured execution times.

The total number of call sites and inlining candidates found during optimization by the WCET-driven greedy inliner are listed in Table 6.3 for every benchmark. The inlining candidates are those call sites that were successfully devirtualized and that passed the various inlining checks presented in Section 4.1. The number of candidates depends on the optimization settings, since inlining more call sites also creates more inlining candidates in the inlined code. Table 6.3 also shows that almost all call sites were successfully devirtualized using the context-insensitive receiver type data-flow analysis, even in the object oriented benchmarks.

Table 6.4 lists the number of call sites inlined in each benchmark, for various optimization settings. For reference, the table also includes the number of

Benchmark	Call Sites			Runtime
	Not resolved	Candidates	Total	Greedy
LineFollower	0	13	26	0.5 s
Kfl	0	122	135	10 s
Lift	0	21	27	1 s
UdpIp	1	55	84	1 s
Debie 1	1	15	73	1.5 s
Debie 2	1	11	40	1 s
Debie 3	0	22	34	0.5 s
Debie 4	0	56	198	3 s
Debie 5	0	107	181	34 s

Table 6.3: Results after WCET-driven inlining

Benchmark	WCET-driven		ACET-driven	
	at most one miss	*always hit*	*at most one miss*	*always hit*
LineFollower	11 / 13	13 / 15	13 / 15	13 / 15
Kfl	17 / 122	40 / 120	21 / 123	59 / 124
Lift	9 / 21	11 / 17	10 / 17	11 / 17
UdpIp	4 / 55	16 / 61	16 / 60	44 / 63
Debie 1	5 / 15	9 / 15	6 / 15	11 / 15
Debie 2	4 / 11	6 / 11	4 / 10	9 / 10
Debie 3	4 / 22	17 / 22	5 / 22	18 / 22
Debie 4	8 / 56	14 / 56	17 / 55	54 / 56
Debie 5	29 / 107	74 / 98	30 / 115	74 / 93

Table 6.4: Inlined call sites / candidates for various optimization settings

inlining candidates, i.e., the number of call sites that passed the initial inlining checks. We can see that the WCET-driven inliner inlines less candidates than the ACET-driven inliner, i.e., the inliner that does not use WCA feedback. The cache cost estimation has a major impact on the number of call sites that are inlined. As we have seen in the execution time results of the jDebie benchmark above, the *at most one miss* approximation leads to a too conservative gain estimation. Only a few of the candidates are inlined when that approximation is used.

The cache approximation that classifies all cache accesses in the all-fit region as *always hit* and all other cache accesses as *always miss* is not included in the results presented here, since in most cases the performance of this approximation is similar to the *always miss* cache approximation, and in the

remaining cases it is similar to the more precise *at most one miss* approximation. There was no benefit over the *always miss* cache approximation for any of the benchmarks.

Also, optimizer configurations that use a top-down or a bottom-up selection strategy are not shown here as there was no real advantage in using those strategies. The top-down strategy creates more inlining candidates since it always inlines callees that have not been optimized before, in contrast to the bottom-up strategy, which inlines candidates in the callee before the callee is inlined at any call site. The top-down strategy may optimize call sites both in the inlined code and in the original callee separately. This requires more updates of the WCA results, which slows the optimizer down slightly. On the other hand, the results of the top-down strategy are similar to the global selection strategy.

The bottom-up strategy results in higher cache miss costs and a higher WCET for three benchmarks. The results are similar to the top-down strategy for the other benchmarks. This suggests that iterating the call graph in a fixed bottom-up order is not a good strategy for architectures that employ a method cache, as it increases the code size primarily at the leaves of the call graph and therefore reduces the size of the all-fit region. The negative impact on the cache costs is larger than the advantage from the fact that bottom-up favors call sites that potentially have the highest execution frequencies.[2] The top-down strategy on the other hand increases cache costs primarily at the top of the call graph, which leads to similar cache costs as the global selection strategy for all benchmarks, with the exception of the Debie 5 benchmark. For this benchmark the top-down strategy results in about 12% lower cache costs as the global strategy. However, as the cache costs contribute to only a small fraction of the WCET bound, the total WCET of the Debie 5 benchmark was only slightly better when the top-down strategy was used, compared to the global selection strategy.

Optimizer Performance and Application Size

Table 6.5 and Figure 6.4 show the total speedup of the WCET that has been achieved by applying both the SimpleInliner and the greedy inliner for every benchmark, as well as the inliner configuration used to achieve the result. It should be noted that although the WCET-driven inliner achieves the best results in most cases, the difference to the ACET-driven inliner is usually very small, as seen in Figure 6.1 and Figure 6.2. JemBench has already been manually optimized, therefore it provides less opportunities for optimization than

[2] It is possible that some leaves of the call graph have a lower execution frequency than their callers. This is the case if all callers of a method invoke that method only in execution contexts with a low execution frequency. However, in the general case the leaves of a call graph will more likely include methods that are called at many call sites with a high execution frequency, such as library methods or JVM methods.

Benchmark	Speedup	Inliner configuration
LineFollower	2.05	any
Kfl	1.09	WCET-driven, *at most one miss*
Lift	1.15	ACET-driven, *at most one miss*
UdpIp	1.01	WCET-driven, *at most one miss*
Debie 1	1.22	WCET-driven, *always miss / always hit*
Debie 2	1.28	WCET-driven, *always hit*
Debie 3	1.54	WCET-driven, *always miss*
Debie 4	1.69	WCET-driven, *always hit*
Debie 5	2.38	WCET/ACET-driven, *always hit*

Table 6.5: Maximum total WCET speedup and the corresponding optimizer configuration

the jDebie benchmark and requires a more conservative cache approximation to avoid a degradation of the WCET, as we have seen before.

The execution times of the WCET-driven greedy optimizer are listed in Table 6.3. They were measured on a PC with an AMD Phenom II processor at 2.8 GHz and 8 GB of RAM. They only include the runtime of the greedy optimizer, and do not account for any other phases of the optimizer. The runtime is primarily determined by the runtime of the WCET analysis used to update the new WCEP after every optimization. If the WCA is disabled, the greedy optimizer executes within a second for all benchmarks.

The rest of the optimizer takes three seconds for loading the application, for the call graph construction and for executing the SimpleInliner, as well as removing unused code and writing the code back to disk. After optimization, the WCA was started separately to analyze the final result, which also took about three to five seconds per benchmark. Executing the whole toolchain, including compiling the Java sources of the benchmarks takes about 15 seconds for a small application like LineFollower, and about 50 seconds for optimizing and analyzing all five tasks of the jDebie benchmark.

The data-flow analysis is not included in those numbers. The results of the data-flow analysis have been pre-calculated for all benchmarks. The execution time of the data-flow analysis ranges from a few seconds for a small application like LineFollower to about 20 minutes for the jDebie benchmark. The inliner transforms the data-flow analysis results during optimization and provides the results to the WCA. Therefore the WCA used to calculate the final WCET bound does not need to run the data-flow analysis a second time. This feature is disabled by default to prevent errors in the data-flow result transformations from influencing the final WCET analysis. However, in the current implementation, passing data-flow results from the inliner to the WCA

Benchmark	Unoptimized	Unused code removed	After inlining
LineFollower	61 kB	25 kB	23 kB
Kfl	64 kB	28 kB	27 kB
Lift	61 kB	26 kB	24 kB
UdpIp	65 kB	28 kB	26 kB
Debie	142 kB	99 kB	97 kB

Table 6.6: Program size before and after optimizations

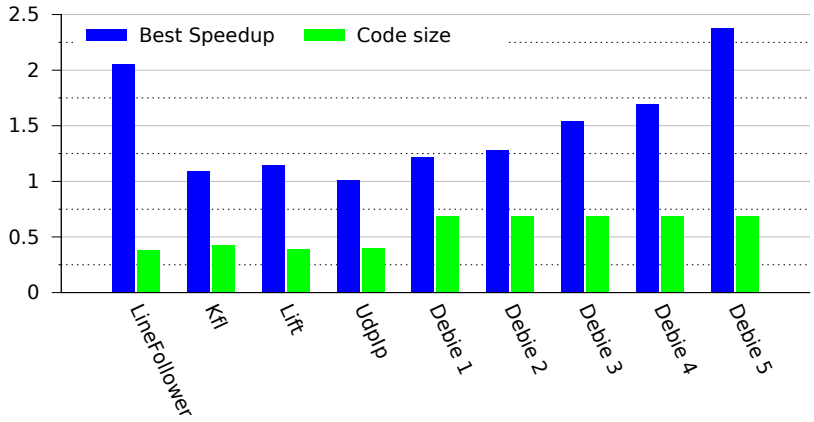


Figure 6.4: Best total speedup and code size reduction compared to the unoptimized benchmarks

does not work if unused code has been removed. In this case, the data-flow analysis needs to be executed once per target method to optimize and one additional time for the final WCA, e.g., a total of six times for the jDebie benchmark.

Table 6.6 lists the size of the generated binary for the benchmark applications before and after unused code has been removed. The program binary includes the bytecode of the methods, information about the class hierarchy and the constant pools, both from the application itself and the part of the JVM that is implemented in Java. Interestingly, the program size decreases after inlining. This is due to the fact that after inlining more methods and classes become unused and are therefore removed. The various optimizer configurations have only little impact on the program size of the benchmark applications. In both the optimized and the unoptimized program binaries, about 60% of the program binaries consist of the bytecode of the methods. The overall code size reduction of the various benchmarks after optimization, compared to the unoptimized code size, is shown in Figure 6.4.

Chapter 7

Conclusion

Modern optimizers are designed to improve the ACET of applications. For real-time applications however, optimizing the WCET is of more interest. WCET-driven optimization is commonly done by focusing the efforts on the WCEP that is found by a WCET analysis. This is complicated by the fact that the WCEP can change during the code optimization.

Many standard ACET optimizations have a positive effect on the WCET if the code size increase is very low. However, on architectures that employ a method cache, optimizations that can lead to a code size increase must be used carefully to avoid increasing the cache costs beyond the gain of the optimization.

To reduce the invocation costs of JOP applications, a WCET-driven bytecode inliner was implemented. To facilitate data exchange between analysis tools and the optimizer, and to avoid redundant code in the toolchain, a new framework was created for the tools. An existing worst-case analysis and an existing data-flow analysis were integrated into a new optimizer using the new framework.

An optimization that does not increase the code size and that does not increase the execution time of any control-flow path never increases the WCET on JOP. Based on this observation a specialized inliner was implemented that is used to inline typical getter, setter and wrapper methods and that does not require feedback from the WCA.

A second inliner was created to handle the more complex case where the code size of the caller increases due to inlining. Feedback from the WCA can be used to optimize only call sites on the WCEP. A cache analysis is used to estimate the gain that is achieved by inlining a call site.

The inliner prioritizes candidates that have a small impact on the application size by selecting candidates with the highest *gain to code size increase ratio* first. However, for the evaluated benchmarks the code size actually decreases after inlining when unused code is removed. The inliner also avoids large code size increases because the size of the methods directly determines

the cache miss costs and hence the gain of the optimizations.

The achieved speedup depends on the benchmark. The small LineFollower benchmark fits completely into the method cache. By inlining nearly all call sites, the WCET bound was cut in half. For the manually optimized JemBench application benchmarks, the greedy inliner reduced the WCET bound by up to 15%. For the more object-oriented jDebie benchmark application, the optimizer was able to achieve speedups from 1.22 up to 2.38, partially due to the good performance of the SimpleInliner.

The benchmark evaluations suggest that a precise cache cost analysis is too conservative in some cases. On the other hand, ignoring the cache costs can lead to optimizations that cause an increase of the WCET in some cases. Focusing on the WCEP during optimization leads to better results than an ACET-oriented optimizer, but it also increases the runtime of the optimizer due to the frequent WCET analyses required to keep the analyzed WCEP up to date, while the faster ACET-driven optimizer also achieves similar WCET improvements in many cases, depending on the benchmark and the cache approximation.

Future Work

There are several ways in which the current optimizer can be improved. First of all, it would be interesting to implement method cloning and method splitting to provide optimizations to reduce the cache costs. Method cloning could be used to create copies of methods for specific call contexts. The copies can then be optimized for the specific call contexts. Method splitting could be used to reduce the size of methods and to move code that is not on the WCEP into a separate method. This would also allow us to generate executable code even if the original application contained methods that are too large for the method cache. Deciding when to use method splitting or method cloning however is not a trivial task and needs support from the method cache analysis. In combination with method inlining, the three optimizations can be seen as a single optimization that re-partitions the supergraph into new methods so that both the method cache costs and the invocation costs are minimized.

To avoid an increase of the WCET during optimization deoptimization should undo optimizations if the WCET analysis calculates a larger WCET after optimization. This would allow us to optimize code more aggressively by using less conservative gain estimations.

It would also be interesting to develop better WCET gain estimations, either by using the WCA to calculate the WCET gain for all candidates in advance, or by calculating the execution time of paths that bypass the optimized code and are therefore potential new worst-case paths after the optimization. However, the final gain of an optimization also depends on future optimiza-

tions that may or may not leverage the optimization potential exposed by the optimization, and a precise WCET gain estimation is very expensive. Therefore an aggressive WCEP-driven optimizer that incorporates deoptimization could provide a better overall performance.

The candidate selector should be improved so that it not only tries to optimize the current WCEP but also other paths that might become new worst-case paths during optimization. Then those other paths may be less likely to prevent the WCET from being decreased by optimizations of the WCEP by becoming the new WCEP.

Interaction with a full-blown WCA is very costly and should be kept to a minimum. Even when a precise WCA is used after every optimization to determine the new WCEP, the actual WCET gain of an optimization is currently only known after the optimization has been applied and analyzed. The estimation of the actual WCET gain for every candidate is either very imprecise or very costly. To the author's knowledge, all current WCET-driven optimization frameworks use feedback from a separate WCET analysis to drive optimization decisions. Integrating the optimization decisions into the IPET-based WCET analysis instead would reduce the number of interactions required between the optimizer and the WCET analysis, while improving the quality of the optimization decisions. The disadvantage of such an approach is that the WCET analysis requires precise knowledge of the code transformations performed by the optimization, and that the resulting ILP problem is probably too large to be solved as whole.

Therefore it would be interesting to explore a third approach, where the precise WCA is used in combination with the data-flow analysis to only provide an initial solution, as well as timing information for basic blocks and flow facts such as loop bounds. Those initial results are used as input for an analysis designed to support the optimizer. A comparatively simple and imprecise high-level WCET analysis that integrates the optimization decision problem might even be more efficient than a very precise WCA driving a simple heuristic optimizer iteratively, especially on target architectures that are very WCET analysis friendly, such as JOP. For example, a specialized WCET analysis could be designed to solve the supergraph repartition problem presented above. Another example would be a WCET analysis that not only provides the worst-case path, but also the execution times of other paths to the optimizer, so that the optimizer could calculate the maximum WCET gain that is achievable by optimizing only a single path or a set of paths.

The optimizer could be improved by implementing a number of standard code optimization techniques that have very little impact on the code size, such as loop interchange, copy propagation, optimization of non-volatile field accesses, or removal of redundant load/store instructions. While those optimizations are not very interesting from a scientific point of view since they are all well known optimizations and do not require interaction with the WCET

analyzer, realizing them will require some work on the optimization framework since currently it does not provide all analyses and data structures required by such optimizations. However, using a separate optimization framework such as Soot to perform non-WCET-critical optimizations instead could remove line number information from the class files, which would prevent the WCET analyzer from reading source code flow fact annotations.

The current optimizer implementation is restricted to a single WCET target method, i.e., it only tries to reduce the WCET for a single root method. Other targets can only be optimized by using either the ACET optimizer without WCA feedback, or by executing the optimizer multiple times for every WCA target method on its output. While it would be possible to restart the WCA-driven optimizations internally for every target method separately, optimizing for a second WCA target method could have a negative impact on the WCET of the previous target method. The current implementation of the WCA does not allow to create multiple WCA instances for different target methods at the same time, which could allow the optimizer to check if a candidate is on the WCEP of any target method, or to prefer candidates which would improve the WCET of more than one WCA target method. A solution could be to extend the WCET analysis to support multiple WCA targets, although this is not a trivial task.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Prentice Hall, second edition, 2006.

[2] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[3] Stefan Hepp. Optimizing Java bytecode for embedded systems. Bachelor's thesis, Vienna University of Technology, 2009.

[4] Benedikt Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[5] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 333–339, Orlando, Florida, May 2008.

[6] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1–2):72–105, June 2010.

[7] Raimund Kirner and Martin Schoeberl. Modeling the function cache for worst-case execution time analysis. In *Proceedings of the 44rd Design Automation Conference (DAC 2007)*, pages 471–476, San Diego, CA, USA, June 2007. ACM.

[8] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot(tm) client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5:7:1–7:32, May 2008.

[9] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium*

on Code Generation and Optimization (CGO'04), pages 75–88. IEEE Computer Society, 2004.

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

[11] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. Automatic WCET reduction by machine learning based heuristics for function inlining. In *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, pages 1–15, Paphos / Cyprus, jan 2009.

[12] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating Pareto optimal compiler optimization sequences—a trade-off between WCET, ACET and code size. *Software: Practice and Experience*, may 2011. DOI 10.1002/spe.1079.

[13] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Martin Maierhofer and M. Anton Ertl. Optimizing stack code. In *Forth-Tagung*, Ludwigshafen, 1997.

[15] Sun Microsystems. JSR 202: Java class file specification update, December 2006.

[16] Oracle. The Java HotSpot performance engine architecture.

[17] Oracle. JSR 336: Java SE 7 release contents, July 2011.

[18] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. A retargetable framework for multi-objective WCET-aware high-level compiler optimizations. In *Proceedings of The 29th IEEE Real-Time Systems Symposium (RTSS) WiP*, pages 49–52, Barcelona / Spain, dec 2008.

[19] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.

[20] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[21] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[22] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[23] 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.

[24] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009. Available at `http://www.jopdesign.com/doc/handbook.pdf`.

[25] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, accepted, 2010.

[26] Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, pages 445–452, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[27] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.

[28] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

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

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

[31] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual

method call resolution for java. In *In Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280. ACM Press, 2000.

[32] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[33] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[34] Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm, Armelle Bonenfant, Hugues Casse, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. Wcet tool challenge 2011: Report. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2011.

[35] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[36] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Syst.*, 34:129–152, October 2006.

[37] Xinrong Zhou, Lu Yan, and Johan Lilius. Function inlining in embedded systems with code size limitation. In *Proceedings of the 3rd international conference on Embedded Software and Systems*, ICESS '07, pages 154–161, Berlin, Heidelberg, 2007. Springer-Verlag.