# CODE COAGULATION IS AN EFFECTIVE COMPILATION TECHNIQUE FOR THE JVM

by

Shruthi Padmanabhan

Bachelor of Engineering in Information Science,

Visvesvaraya Technological University,

2009

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2018

## Author's Declaration

## Abstract

CODE COAGULATION IS AN EFFECTIVE COMPILATION TECHNIQUE FOR THE JVM

Shruthi Padmanabhan
Master of Science, Computer Science
Ryerson University, 2018

Compilers are the interface between programmers and computers. The output of a compiler determines the speed of the resulting program as well as its energy footprint – of increasing importance in this world of climate change and battery-dependent computing. Code coagulation has the potential to improve that performance, particularly relative to (JIT) interpreters. Java and related languages are in wide use, so addressing their performance has large potential impact. Optijava is an experimental compiler that applies code coagulation to Java programs.

## Acknowledgements

I would first like to thank my supervisor Dr. Dave Mason, for many insightful conversations, encouragement and helpful comments in this thesis. His advice has been invaluable. He has been a pillar of support and allowed me to grow as a research student.

I would also like to thank my committee members, Dr. Marcus Santos, Dr. Isaac Woungang and professor Sophie Quigley for serving as my committee members and letting my defense be an enjoyable moment, and for suggestions and comments.

Finally, my sincere gratitude to my family, my husband Krishnaprasad Athikkal, my lovely daughter Ameya Krishnan, parents, and in-laws for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

My thesis is that Code Coagulation is an effective compilation technique for the JVM. The strategy of code coagulation is local optimality. It starts by compiling frequently executed parts of the code and locally allocates optimal resources to them. Resource adjustments are done - as required - on the less frequently executed paths through the code. This produces highly efficient code in terms of instruction selection and register allocation.

## 1.1 Problem Background

The Java Virtual Machine (JVM) is the core of Java technology. The JVM is an abstract machine that executes the Java bytecode. The JVM delivers the pivotal feature of Java; platform independence. The JVM initially consisted of an interpreter which sequentially executed the bytecode instructions [36]. However, this negatively affected the performance since the interpretation od each instruction would take several CPU instructions and many cycles. To address this JVMs started to incorporate a Just In Time (JIT) compiler to improve the performance [4]. During program execution, a JIT compiles a Java method to native code "on the fly", and saves the compiled version on the heap. On future references to the same method, the compiled method is executed directly without the need for interpretation. The compilation process of JIT includes flow analysis and register allocation algorithms to generate optimized and higher-quality native code. This has helped to produce efficient machine code with significant performance improvement.

However, CPU time, speed and performance will always be an issue in long-running applications such as web servers. Any JIT will consume a lot of CPU time at startup, and for its optimizations. Since the program execution includes the JIT compilation time, it affects the overall speed and CPU time. Hence, it is important that the compiler generate faster code to reduce the application execution time. The compiler should also be able to use the resources efficiently by consuming less CPU time and thereby conserving energy.

JVMs have evolved over the years incorporating optimizating compilation techniques to improve execution speed. State of the art JVMs use adaptive compilation. The adaptive compiler helps in identifying performance critical methods or methods that contains loops or any performance critical paths and optimizes these methods as opposed to optimizing all the methods as done by conventional JIT. This helps to reduce the overall compilation time. Nonetheless the startup speed of the compiler remains an issue [35].

What would be useful is a simple compiler which has less overhead, an effective optimization technique and which helps in faster execution. The alternative is an upfront compiler (a compiler that translates the program into native code up front, well before the program is even run) that does not add overhead to execution. We propose an optimizing Java compiler, OptiJava; a simple upfront compiler written in Java which aims at a speedy execution of Java programs. OptiJava uses a code generation approach called code coagulation. Code coagulation is an optimization technique that aids in efficient resource utilization [21].

## 1.2 Objectives and Proposed Methodology

Optijava is an upfront, feedback-directed, compiler. Up front: OptiJava compiles the Java class files to native code before run-time. Feedback-directed: Optijava improves the performance of a program based on branch-frequency information gathered at run-time from previous executions.

The generated native code is run on a typical workload and the class files are recompiled using the gathered frequency information (number of times each part of the code was executed). With this information, it can make context sensitive decisions.

The overview of OptiJava is as shown in Figure 1.1.

OptiJava uses a compile only approach, i.e. , it does not use an interpreter. The class files are directly fed to OptiJava without any interpretation. A distinguishing feature of this compiler is that it uses a different approach to code generation namely code coagulation. This approach is a cost-effective method to utilize resources [26]. In this approach, the frequently executed code known as hot-spots are compiled with a locally optimal use of registers and instructions, and then the less frequently executed code is

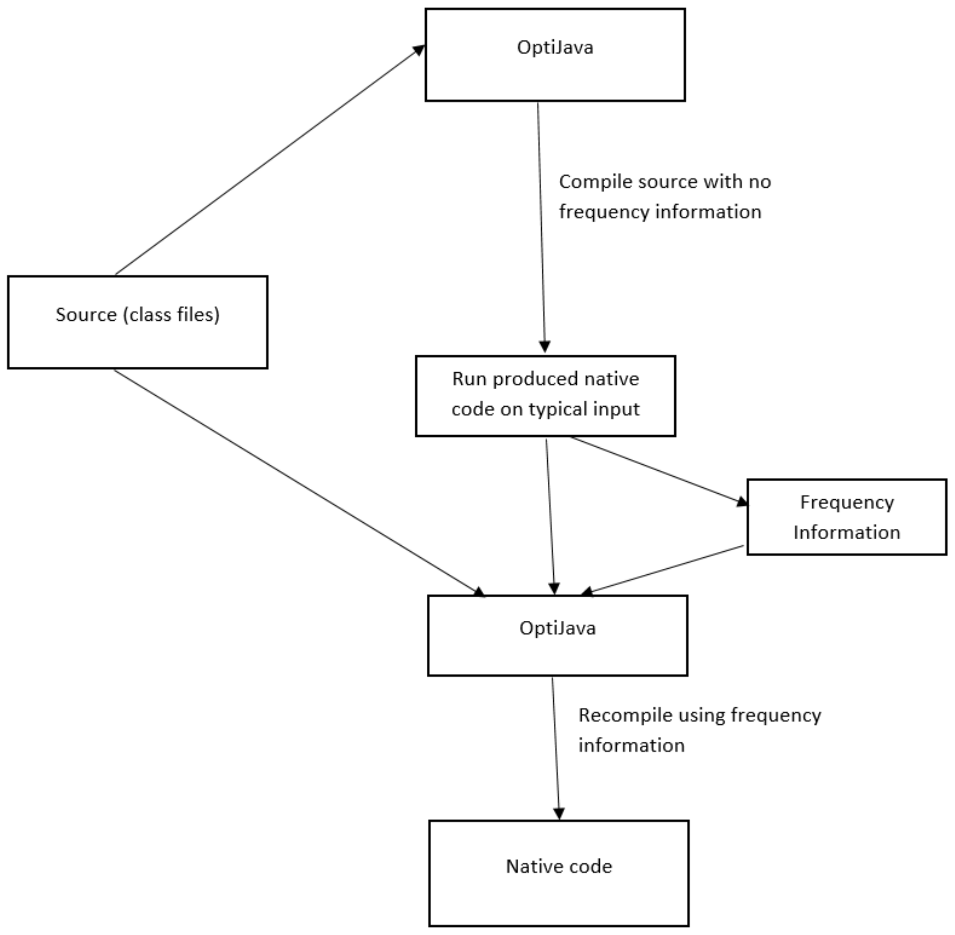

OptiJava
Compile source with no frequency information
Source (class files)
Run produced native code on typical input
Frequency Information
OptiJava
Recompile using frequency information
Native code

Figure 1.1: Overview of OptiJava compiler

compiled. This will be explained in more detail in Section 3.5. Code coagulation was proved to be an optimal solution in C compiler [26]. We propose to use this strategy in Java compilation.

OptiJava initially loads all the required classes and methods. Methods are then partitioned into basic blocks and a control-flow graph is generated which is used by the coagulation phase. Coagulation in itself has two phases: unification and solidification – explained in detail in Section 3.5.4 and Section 3.6 respectively. After coagulation, the basic blocks are given a sequence that is optimized for branches and cache references. Finally, the instructions are converted to assembly instructions.

OptiJava is designed to support multiple architectures (including IA32, PPC, and ARM), but currently the primary target is AMD64 (colloquially known as x86-64). Our ultimate goal is to implement this compiler in long-running application servers to produce faster code thereby assisting in energy saving and carbon reduction.

## 1.3 Contributions

The main contributions in this dissertation are:

- Implementation of the coagualtion concept in an abstract virtual machine.
- In the past, the coagulation technique has only been implemented in a C compiler. Through this research, we have implemented the coagulation concept in an object oriented language for the first time.
- Generally, compilers convert intermediate instructions to be in a Static Single Assignment (SSA) form. This process of conversion is time consuming. We have introduced a technique where the intermediate instructions can be of SSA form implicitly without having to carry any additional conversion process.

## 1.4 Dissertation Outline

This dissertation is organized as follows:

- Chapter 2 presents some background on the Java Virtual Machine and the execution techniques used in the JVM. It also includes a basic overview of the structure of a modern compiler. Related work in Ahead-Of-Time compilation is also discussed.
- Chapter 3 discusses the design of OptiJava in detail.

- Chapter 4 analyzes the performance of OptiJava. Analysis is done on the basis of a program containing few loops. The run-time statistics obtained from OptiJava and that of a commercial JVM are compared.
- Chapter 5 concludes by presenting conclusions and suggestions for future work.

# Chapter 2

# Related Work

Java's "write once/run everywhere" concept has contributed to its mainstream success. This was made possible by the Java Virtual Machine (JVM). Java programs are compiled into platform independent java byte code (`.class` files). JVM, an abstract machine, translates and executes the byte code. Unlike C, Java programs need not be recompiled for different architectures because the JVM takes care of the final transition to native code. "All platforms on which a JVM exists can execute Java" [36].

## 2.1 Java Virtual Machine

"Java Virtual Machine is an abstract machine that executes a Java program and is the key to many of Java's features, including its portability, efficiency and security" [36]. An implementation of the JVM should have the features that are specified by the JVM specification [23].

The JVM dynamically loads, links and initializes classes and interfaces when they are needed. Loading is the process of locating the binary representation of the class and creating a class or interface structure from that binary representation. Linking transforms the loaded class into a runtime representation of the JVM. Once the classes have been loaded and linked, the classes are ready for execution. The bytecodes of the class are executed by the execution engine. A simplest form of an execution engine is a bytecode interpreter which interprets the bytecodes one at a time. Other ways of execution are Just-In-Time compilation and Ahead-Of-Time compilation. The execution techniques are explained in detail in Section 2.2.

### 2.1.1 Java Virtual Machine Architecture

The JVM can be viewed as a collection of subsystems, run time data areas and execution engine. A block diagram of the JVM is as shown in Figure 2.1 (from [36]).

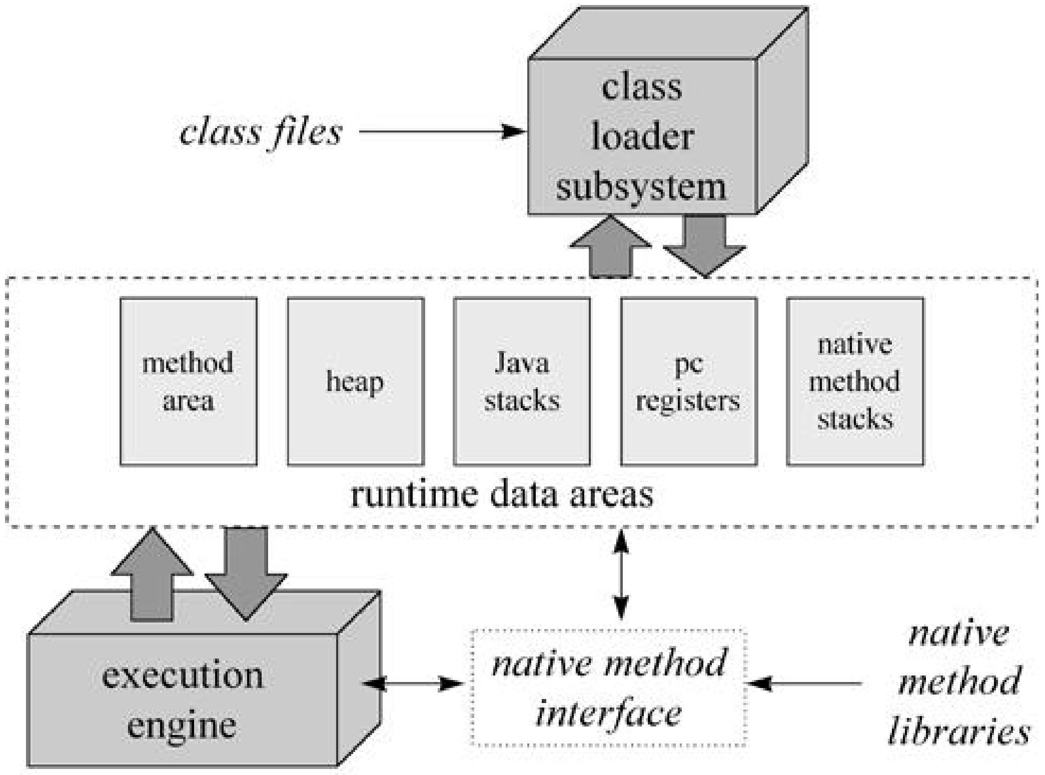


Figure 2.1: JVM architecture

**Class Loader Subsystem**
Class Loader subsystem is responsible for loading the classes and the interfaces dynamically. It loads the class files from both the program and the Java API (set of runtime libraries). Class files are located, linked and initialized by this sub system.

The process of obtaining the Java class file (binary representation) and placing them in memory is known as loading. A class is said to be loaded when its binary representation is loaded. A class file contains vital information such as methods of the class, symbolic reference to its superclass and constant pool. After loading, the JVM has knowledge about the name of the class, its hierarchy, the fields and methods of the class. Once the class has been loaded, the loaded representation is verified for its validity and its conformance with the security constraints. Then memory is allocated for class variables and the variables are initialized to default initial values. Next the symbolic references in the class are transformed to direct references. Finally, the class variables are initialized to their actual initial values as described in the program by invoking the static initializer `<clinit>` containing all the class variable initializers and static initializers of the class.[36]

**Run Time Data Areas**

The JVM requires memory to store bytecodes, information extracted from class files, objects, parameters, return values, local variables and so on. These are stored in runtime data areas. Run time data areas consist of Method Area, Heap, Java Stack, Program Counter and Native Method Stacks.

Class data, special methods used in class initialization, method data and constant pool (an ordered set of constants, literals, symbolic references to types, fields and methods) are stored in `method area`. The memory for new objects and arrays are allocated from the `heap`. Each object in the heap has an associated class in the method area. The `Java stack` holds a method's state invocation such as method's local variables, parameters, return value and partial results. The Java specification supports the use of native code through the Java Native Interface (JNI) specification, where external libraries (DLLs or shared objects) are dynamically loaded and then wrapped with Java classes. A `native method stack` is used by a native method to store its state. The address of the current instruction of the method being executed is stored in `Program Counter` (PC) register [36].

### 2.1.2 Java Stack

The JVM is a stack-based machine that continuously fetches and decodes bytecodes. A method's data operations, partial results, return values, dynamic linking are done via a stack. The Java stack holds a method's state invocation such as local variables, parameters and return value.

The Java stack consists of a stack frame for each method. A stack frame has an operand stack, local variables and frame data.

**Local Variables**

Local variables of an invoked method are stored as an array of variables. The length of the array is determined at compile time and is provided by the class file. The variables in the array are accessed by indexing. Indexing starts from 0. If the method is an instance method, then index 0 is used for storing the reference to the object containing the invoked method (this) and the local variables will start at index 1.

#### Operand Stack

Operand stack is a last-in-first-out stack. Instructions take their operands from the stack, operate on them and pushes the result on to the stack. The stack is also used to pass the method arguments and to receive the result.

Consider the following example of an add method in Java source code and then in Java bytecode format:

```
public int add(int a,int b){
return a + b;
}

iload_1   // push the int in local variable 1
iload_2   // push the int in local variable 2
iadd      // pop two ints, add them, push result
ireturn
```

In this sequence of bytecodes, the first two instructions, `iload_1` and `iload_2`, push the integers stored in local variable positions one and two onto the operand stack. The `iadd` instruction pops those two int values, adds them, and pushes the int result to the operand stack. The fourth instruction, `ireturn`, pops the result off the top of the operand stack and terminates the method using the result as the return value [36].

#### Frame Data

Frame data consists of the data that performs dynamic linking, constant pool resolution, return values for methods, and dispatch exceptions.

### 2.1.3 Execution Engine

The core of the JVM is its execution engine. It takes a sequence of bytecodes of the loaded method and executes the instructions that the bytecodes represent. A runtime instance of an execution engine may execute bytecodes directly by interpreting or indirectly by compiling and executing the resulting native code.

#### Instruction Set

The bytecodes of a method are a sequence of instructions. Each instruction consists of a one-byte opcode followed by zero or more operands. The operation to be performed is indicated by the opcode. The operands will be the entries from the constant pool or entries from the operand stack. Execution is done one instruction at a time. Some opcodes have a prefix which indicates the type they operate upon. For example, an `add` instruction has two implementations; `iadd` indicates that the addition is done on two integers and `fadd` indicates that the addition is done on two floats. The prefix informs the virtual machine whether it has to perform an integer or a floating-point arithmetic. Some opcodes do not have a prefix, indicating that it can be performed on any type of data. Opcodes such as `pop`, `dup`, `invokevirtual` are not associated with any specific type.

The execution engine fetches an opcode and its operands, if any. It then executes the operation as indicated by the opcode. The next opcode to be executed is usually the one following the current opcode. However, for instructions such as goto or return, the opcode might be several instructions away. This process of fetching the opcode and executing the operations continues until the method has completed i.e. returned.

If an instruction throws an exception, i.e, if during the execution of the instruction an event occurs that disrupts the normal flow of the program, then the execution engine handles the exception by determining an appropriate catch clause in the exception table. If there is no appropriate catch clause, the exception is passed to the calling method for handling.

If the execution engine comes across a native method invocation, then, the engine invokes the native method. The invocation instruction is executed by running the native method and the engine continues with the next instruction when the native method returns [36].

## 2.2 Execution Techniques for the JVM

The different ways of executing the bytecodes by the execution engine are : Interpretation, Just-In-Time compilation and Ahead-Of-Time compilation.

### 2.2.1 Bytecode Interpreter

A Java bytecode interpreter is the original and the most commonly used execution technique for a JVM. The JVM interpreter executes the instructions sequentially. The interpretation of a bytecode instruction involves fetching the instruction, decoding it and performing the operation [13]. A typical bytecode interpreter is structured in a loop and is implemented in C. Interpreters are simple and easy to implement and help in the security and portability of the language. However, it is very slow [11].

Instruction dispatch i.e., fetching the instruction and decoding the instruction consumes most of the interpreter's runtime. The JVM interpreter must fetch, decode, and then execute each bytecode one by one. Consider the following expression:

```
x = y + ( 2 * z )
```

The bytecode sequences of this instruction would be:

```
iload_2       //push y (local variable 2)
iconst_2      //push constant 2
iload_3       //push z (local variable 3)
imul          //multiply 2 and z
iadd          //adds y and the result of the previous computation
istore_1      //store the result to x (local variable 1)
```

Evaluating this expression involves decoding the six bytecode sequences and performing the operations specified by the bytecode sequences. Determining the semantics of each individual bytecode and performing the appropriate computation involves processor and memory usage. This affects the run time performance of the Java application [11].

Apart from dispatching, the performance of the interpreter is also affected by the number of memory accesses. Interpreter for the stack based JVM frequently access the operand stack in memory to push and pop its stack operands. Frequent load and store operations such as fetching the bytecode from the memory or storing the result to the memory poses a bottleneck for performance.

### 2.2.2 Just In Time Compiler

To improve the performance, bytecodes can be compiled into efficient instruction sequences for the underlying machine [11]. Compiling at run-time is the essence of a Just In Time (JIT) compiler. A JIT compiler interacts with the JVM at run time and compile appropriate bytecode sequences into native machine code. The translated code is cached to eliminate the repeated translation of the same sequence of bytecodes. This is in contrast to an interpreter, an interpreter must translate a bytecode sequence each time it is encountered. For example, if a program has a loop or a recursion, the interpreter translates the bytecodes repeatedly. However, in the JIT compiler the translation of bytecodes is done only once and the translated code gets executed each time.

Traditionally a Java JIT compiler translates the bytecode into native code when a new method is invoked during run-time. When the method gets invoked for the first time, the JIT compiler compiles the whole method to native code just before execution. The compiler first converts the method's bytecodes to an intermediate representation which represents the machine code more closely. The intermediate representation is analyzed, optimized and then translated to native code [11]. Once the method has been compiled, during the next invocation of the same method, the JVM calls the compiled method instead of interpreting it.

However, this approach of JIT compilation significantly affected the startup time of a Java application. When the JVM first starts up, thousands of methods are called. Compiling these methods at run-time can significantly affect the startup time. Ding et al. [12] and Radhakrishnan et al. [31] observed from their study that the JIT compiler with this traditional approach did not provide any significant improvement in the performance. They observed that the JIT compiler will have better performance than an interpreter only if the compiled method was used more frequently. To improve the startup time and performance, the JIT compilers were modified. Instead of compiling method by method, frequently executed methods or methods containing frequently executed part of code known as “hot spots” were converted to native code [20] . This technique is used in Sun's HotSpot JVM [35] and in the IBM's J9 JVM [34].

Sun's (Oracle's) HotSpot JVM uses a mixed mode of compilation where the interpreter first runs the bytecodes and it detects the “hot spots” while it runs. The JIT compiler gets invoked when it detects frequently executed methods or frequently executed parts of the code. The JIT compiler then compiles the frequently executed methods and the methods that contains the “hot spots”. Future references to the same methods points to the compiled methods. This avoids unnecessary compilation of infrequently executed code [28].

Similar to Sun's HotSpot JVM, in J9, a JVM by IBM [34], the JIT compiler is enabled by default and is activated when a method is called. Instead of compiling all the methods at start up, only the “hot spot” methods are compiled. That is, the methods which are called frequently are compiled first. JVM maintains a call count for each method, the count gets incremented each time the method gets called. The interpreter executes the method until the count reaches a certain threshold. When the count reaches the threshold, the method gets compiled by the JIT compiler. The next call to this method will now call the compiled method. Thus, frequently used methods get compiled at start up and the less used methods gets compiled much later or might not get compiled at all[34].

Eventhough, the performance of a Java application is significantly improved using the JIT compilation, but there could be a potential performance penalty caused by the infrequently-used methods. Since infrequently used methods are not compiled, there will be repeated interpretation of these methods. Another downside of JIT compilation is that a Java program's execution time will now also include the compilation overhead of the JIT compiler.

### 2.2.3 Ahead Of Time Compiler

Another approach to execute the Java bytecode is an Ahead-Of-Time (AOT) compilation, also known as upfront compilation. This compilation approach translates all the Java bytecode sequences to native code before it is run. Since the translation is done before execution, the run time overhead of the JVM is reduced thereby improving the start-up time of both small and large Java applications[36].

AOT compilation can be classified as native and non-native. Native compilers directly produce executable code from the Java bytecode and non-native compilers produce a C code from the Java bytecode and the C code is translated and executed by an existing C compiler.

An AOT compiler can be implemented in two ways: either as a standalone executable that compiles and executes the program by providing the necessary run time services or by generating native code that is interfaced with the JVM. In either of the implementations, the source is translated into some intermediate form. Various analysis and optimizations are performed on this intermediate form. The translation and optimization is done offline, before the bytecode is loaded for execution. The advantage is that the compiler now has time to fully analyze and optimize the bytecode stream. This has the potential to yield native code with better performance than code produced by a JIT compiler.

Java's "Write Once,Run Anywhere" paradigm is supported by providing static Java compilers targetting all major platforms, just like it is supported right now by providing a JVM for each of them. The disadvantage of this approach is that it does not support dynamic loading because Ahead-of-Time compilation requires pre-compilation of all the classes that the application may use.

## 2.3 Compiler Structure

The basic structure of a modern compiler is fairly standardized [1]. The stages of scanning, parsing, and semantic analysis are outside the scope of this work, as we are based on Java class files which have already been compiled to the JVM. However subsequent stages exist in the OptiJava compiler.

Usually, a compiler does not translate a program in a high-level language directly to machine code, but instead first translates it to a program in a slightly smaller, simpler language. These simpler versions of the original are known as the Intermediate Representation (IR) of the program. Types may also be applied to these simpler intermediate representations further strengthening the implementation. An IR is any data structure that can represent the program without loss of information so that its execution can be conducted accurately. Since its use is internal to a compiler, each compiler is free to define the form and details of its IR.

Code generation, an important phase of a compiler, takes the IR as input and produces a semantically equivalent target program as its output. The target program generated must preserve the semantics of the source program and must make effective usage of the available resources (registers, memory) [1].

Code generation essentially consists of:

- Identification of Basic Blocks
- Dataflow Analysis
- Instruction selection
- Register allocation and assignment
- Instruction ordering

### 2.3.1 Basic Block and Control Flow Graph

The simplest unit of control flow in a program is a basic block. The IR instructions are partioned into basic blocks. A basic block is a linear sequence of instructions that contains no branches except at its very end. A basic block has the following properties:

1. The flow of control can enter a basic block only through its first instruction.
2. Control will leave the block without halting or branching except at the last instruction of the block.

Instructions within a basic block are always executed sequentially as a unit.

A control-flow graph models the possible run-time flow paths. It represents the flow of control between the basic blocks. It is a directed graph, where each node corresponds to a basic block and each edge corresponds to a possible transfer of control between the blocks [3].

### 2.3.2 Data-flow Analysis

Translating the source code to native code naïvely can introduce substantial run time overhead. Hence compilers perform optimizations to reduce the run time overhead. The goal of code optimization is to discover, at compile time, information about the runtime behavior of the program and to use that information to improve the code generated by the compiler. The most common goal of optimization is to make the compiled code run faster [3]. Code optimizations gathers the information at compile time by data-flow analysis.

Data-flow analysis determines how the data flows throughout the program. It allows the compiler to evaluate the runtime flow of values in the program at compile time. It generally uses the control-flow graph, to understand the flow of data. Data-flow analysis views computation of data through expressions and transition of data through assignments to variables.

Data-flow analysis are classified as follows:

- Local data-flow analysis : Analysis across statements but confined to a basic block.
- Global(Intra-procedural) data-flow analysis : Analysis across basic blocks but confined to a method.
- Inter-procedural analysis : Analysis across methods.

The information gathered from data-flow analysis, is used by instruction scheduling and register allocation. Further, various classical optimizations can be performed using data-flow analysis, such as, common subexpression elimination (eliminating redundant expressions), constant propagation (substituting the values of known constants in expressions at compile time), dead code elimination (removal of unreachable code/removal of code that does not affect the behaviour of the program), etc.[33].

#### Static Single Assignment

Data-flow analysis finds the uses of each defined variable or definitions of each variable used in an expression. For instance, in the expression

```
x = y + 1;
```

the variable x is being defined/assigned and the variable y is being used.

Data flow analysis is much simpler when the variable is defined only once. This forms the the crux of the Static Single Assignment (SSA) form. A program is said to be in SSA form if each variable is assigned exactly once in the program. Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. The compiler modifies the intermediate representation of the program to be in SSA form.

Converting the IR to a SSA form involves renaming of variables that are targets of more than one definition. That is, every time a variable is defined (assigned) in the code, a new version of the variable is created. Only one definition reaches every usage and this makes optimization algorithms simpler, precise and efficient. It has proven useful in both analysis and transformation and has become a standard representation used in both research and production compilers [33].

The intermediate instructions in OptiJava are of SSA form. However, unlike other compilers which convert their instructions to SSA form, the intermediate instructions in OptiJava are implicitly in SSA. This is explained in detail in Section 3.4.

### 2.3.3 Instruction Selection

The process of selecting appropriate target machine instructions to implement the IR statements is known as Instruction selection. Instruction selection maps IR statements into semantically equivalent instructions of the target processor. It is important to note that the target machine instructions should have the same semantics as the IR statements. A naïve translation, that is, translating IR statements one by one may result in correct result but it would result in a less efficient target code. There may be redundant load and stores which could have been avoided. Consider an instruction

```
a = a + 1;
```

If the target machine supported an increment instruction, then instead of using three statements; loading the variable a , adding 1 to the variable, storing the result; the instruction can be implemented more efficiently using one single increment statement [1].

### 2.3.4 Instruction Scheduling

Instruction scheduling is the reordering of the instructions to reduce the total number of processor cycles required to execute an instruction. Instruction scheduling can be done locally by reordering the instructions within a basic block or it can be done globally by reordering the instructions across the basic blocks. The reordering of the instructions must preserve the data dependencies and the semantics of the program. Picking the best order of instructions is an NP complete problem [1].

The rationale of instruction selection and instruction scheduling has changed in the modern architectures. Now the focus of instruction selection and instruction scheduling is to produce smaller and denser code. In this way more code will fit in the cache and execution will be faster.

### 2.3.5 Register Allocation

Instructions involving register operands are invariably shorter and faster than those involving operands in memory. Registers are the fastest computational unit but unfortunately the target machines will not have enough registers to hold all the values. So, efficient utilization of registers is important. Register allocation involves determining how many registers are required and allocating registers symbolically. Assignment involves determining which of the actual hardware registers will be used for each allocated register. Finding an optimal assignment of registers to variables is an arduous task and is an NP complete problem [1].

The widely used register allocation algorithms are:

- Graph coloring algorithm
- Linear scan algorithm

Both graph coloring and linear scan allocators use liveness information for register assignment. Liveness information determines whether the variable is live at a given point of the program, i.e. it contains a value that may be used at a later point in the program. Liveness analysis plays a critical role in register allocation. “The register allocator need not keep values in registers unless they are live; when a value makes the transition from being live to being not live, the allocator can reuse its register for another purpose” [3]. The liveness information is gathered using data-flow analysis.

Graph coloring allocators encapsulate the liveness information of the variables as an interference graph. Each node in the interference graph represents a variable. An edge connects two nodes if the variables represented by the nodes interfere, i.e., they are live at the same time and cannot be allocated to the same register. “If the machine has K number of registers and if it is possible to K color the graph – i.e. color the graph with K colors – then the coloring is a valid register assignment. For a k-register target machine, finding a k-coloring of the interference graph is equivalent to assigning the candidates to registers without conflict” [8]. A traditional graph coloring allocator builds an interference graph and heuristically attempts to color it. If the heuristic succeeds, the coloring results in a register assignment. If it fails, some register candidates are spilled to memory, spill code is inserted for their occurrences, and the whole process repeats [8].

Linear scan allocators view liveness information as a lifetime interval. The live interval of a variable 'v' is the set of 'm' to 'n' instructions, where 'm' is the first instruction at which 'v' is first defined and 'n' is the last instruction at which it is last uesd. A variable is considered as dead at all instructions outside the live interval. The allocator can use this information to easily determine how these intervals overlap and assign variables with overlapping intervals to different registers. The linear scan algorithm first arranges all the instructions of a method in a linear order. Then, lifetime intervals for all the variables are computed. The linear scan algorithm operates directly on the list of intervals, sorted by their start positions. The compiler iterates over the list and assigns a physical register to the interval immediately. If no physical register is available for the whole lifetime, then some intervals will be spilled to memory. Two lifetime intervals interfere if their ranges intersect. So, two variables whose lifetime intervals do not intersect, will be assigned with the same physical register [29].

### 2.3.6 Code Generation Techniques

Instruction selection, instruction scheduling and register allocation are part of the code generation phase of a compiler. Every processor uses an instruction pipeline. With an instruction A common technique used in code generation is to perform instruction scheduling before register allocation. The order in which the instructions are arranged for execution has a significant effect on the time it takes to execute a sequence of instructions. Instruction scheduling gives priority to the number of instructions that can be executed parallely (instruction-level parallelism) which minimizes the total number of processor cycles required to execute instructions. The instructions that are not dependent on each other can be However, executing instructions in parallel creates the need for more registers to hold the values being computed simultaneously. Scheduling instructions also may increase the register lifetimes if it increases the time between a write to a register and last read of that value. Longer lifetimes increase the number of concurrent live registers thereby increasing the contention for registers and increasing the chances of register spills (storing and restoring value of a register to/from memory) [3].

The conventional alternative approach is to perform register allocation before instruction scheduling. This gives priority to utilizing registers over exploiting instruction-level parallelism. This approach was initially proposed by Hennessy[16]. It was a common approach used in early compilers when the target machine had only a small number of available registers. But this approach may affect instruction scheduling, because the register allocation could inadvertently introduce dependencies by allocating the same register for unrelated instructions.

The separation of the register allocation and instruction scheduling phases leads to significant problems, such as, poor optimization and additional complexities in trying to adjust one phase to consider cost considerations from the other [1]. Hence, in general the compiler writer faces the problem of determining which phase should run first to generate most efficient code. An efficient solution to this problem would be to integrate register allocation and instruction scheduling [6].

### 2.3.7 Code Coagulation

Generating an optimal target program is hard. Hence, heuristics plays a major role in code generation. A carefully designed code generator can produce code that is several times faster than the code produced by a naïve one [3].

Karr [21] proposed a code generation design, named code coagulation, which integrated the register allocation and instruction scheduling. In this approach the instruction selection and register allocation are always done together. Also, it treated the busy parts of the program first when registers are abundant. This resulted in producing a native code that was highly efficient. Morris [26] designed a Coagulating Code Generator (CCG) which used Karr's[21] approach of code generation. A study conducted by Karr, Morris, and Rozen (1991) demonstrated that there is substantial speedup by CCG over GNU C compiler and it was found that CCG generated highly efficient code in terms of instruction selection, register usage, and procedure calls [22].

The basic idea of code coagulation is to optimize and compile small regions of code locally in isolation and then merge the compiled parts of the program. But the order in which the regions are merged together matters. The regions that are of higher importance are merged first, i.e, the frequently executed regions of code are merged first. "The merging is done in decreasing order of execution frequency, the idea being that the instruction selection and register allocation is properly arranged on the expensive paths through the program. For example, inner loops will be compiled first, registers arranged, etc. But those pieces inside loop that are seldom used will have no influence on the initial register assignments.[21].

The procedure followed in coagulation is as follows:

1. A control flow graph is built where each node represents a basic block and each edge represents the flow of control between the blocks.

2. A profile is used to label each edge of the graph with its expected execution frequency. Initially, all the edges are marked as uncompiled.

3. Instructions are selected and register allocation is done for each node in isolation at minimal cost. Now, each node will have its own boundary conditions regarding the location and the data it uses and supplies.

4. An uncompiled edge with the highest frequency is selected. If the boundary conditions on the edges entry and exit nodes do not agree then a minimal cost repair such as inserting a copy instruction or revising storage allocation is done. Otherwise, the edge is marked as 'compiled' and the nodes are merged.

5. Step 4 is repeated until all the edges are compiled.

Thus, in effect, instead of compiling at method-level, coagulation technique compiles at basic block level, where register allocation and instruction selection are simultaneously done to reduce the cost.

OptiJava uses code coagulation as its compilation technique. Karr's proposal separated the isolated compilation of the regions and the selection of edges. That is, Karr proposed to first compile all the small regions and then coagulate the edges. However, unlike Karr's proposal we combine the selection of edge and compilation as a single step. Highest frequency edge is coagulated and then any uncompiled region at the either end of the edge is compiled.

## 2.4 Related work in upfront Java compilation

Over the years, several Java upfront/AOT compilers have been developed. This section reviews the work done by other researchers in the area of AOT compilation for Java.

Toba [30] is a bytecode to C compiler. It compiles the Java bytecode to C and relies on a C compiler to translate the C code to native machine code. Toba consists a bytecode-to-C translator, a garbage collector, a threads package, a run-time library, and native routines implementing the Java API. Toba translates each Java method into a C function, and these functions share a global namespace. Overloaded methods are distinguished by using a suffix, that encodes the class name, the method name, and the method signature All reference types are translated into a C pointer type. It maps each JVM stack location to a C variable. The bytecode is converted to an Intermediate Representation (IR). The IR is then transformed to equivalent C code. For stand-alone applications that do not rely on dynamic loading, Toba provides large performance benefits. Toba is no longer maintained or supported [30].

TurboJ [37] is another bytecode to C compiler. It improves upon Toba by providing support for dynamic loading by having a mixed mode of execution. It converts the java bytecode to C before execution and during execution it interfaces with JVM to utilize the thread management, memory and class and library loading services. Thus, TurboJ is not a stand-alone Java runtime system. Instead it operates in conjunction with a Java runtime system and uses the native JDK on a given platform [37].

Bytecode to C compilers focus on optimizing the intermediate representation, because the final C code will be optimized by the C compilers of the platform. Since C compilers are available in most of the platforms, the Java's potability feature is supported. However, a lot of Java specific information is lost during transformation which is useful for optimization. Thus the resultant executables still have low performance. There is also a significant overhead in converting Java methods to C functions and there is inefficient usage of registers [19].

Harissa [27] is a Java environment that improves upon the drawback of previous compilers by using a mixed mode execution technique. It includes both a bytecode to C compiler and an interpreter integrated into the runtime library. It currently has support for SunOS, Solaris, Linux, and DEC Alpha platforms. "Harissa's compiler takes as input a Java class containing a main method and generates as output a makefile, a main.c file, and a C source file for each class used in the program. To determine the set of classes that depend on the initial class, an analysis is recursively performed on the byte-code to search for all the classes referenced by the main class [27]". The Harissa compiler reads in the bytecode and converts it into an Intermediate Representation (IR). It then performs Java specific optimizations on IR such as method inlining, eliminating type checking and array bound checking on array indices that can be statically detremined. A complete interpreting JVM has been integrated into the runtime library to allow dynamic loading. Since data structures are compatible between the compiled code and the interpreter, Harissa provides an environment that cleanly allows the mixing of bytecodes and compiled code [27].

Caffeine [18] is a Java bytecode-to-native code compiler that generates optimized machine code for the x86 architecture. The compilation process involves several translation steps. First, it translates the bytecodes into an internal language representation, called Java IR. Next using stack analysis, class hierarchy analysis and stack to register mapping the Java IR is then converted to a machine-independent IR, called Lcode. The Lcode is then optimized by applying optimizations such as inlining, data-dependence and interclass analysis. Later, peephole optimization, instruction scheduling, and register allocation are applied to convert Lcode to machine-specific IR. Finally, optimized native code is generated from this machine-specific IR. Caffeine uses an enhanced memory model to reduce the overhead due to additional indirections specified in the standard

Java memory model. It does not support garbage collection, threads, and the use of graphics libraries [18].

GNU Java Compiler (GCJ) [5] is an AOT compiler which can compile both the Java source code and the Java bytecode to native code. GCJ requires a runtime library that needs to be ported for each architecture. An advantage of GCJ is the comparatively faster startup speed and modest memory usage. The stack based bytecode is translated to an intermediate level representation by creating a virtual register for each of the Java local variables or the Java stack values. GCJ later assigns a hardware register or stack location for each of the virtual registers. GCJ makes use of all the optimizations and tools already built for the GNU tools. The optimized intermediate representation gets converted to assembly. The GNU assembler processes the created assembly file and the resulting object file is linked into an executable [5]. As of GNU Compiler Collection 7(GCC), GCJ is no longer maintained and has been removed from GCC [15].

Marmot [14] is a standalone compiler. Marmot consists of an Ahead-Of-Time compiler, run time systems and libraries. Marmot system converts the bytecodes to a SSA based IR and then converts it into native code. Compiling a Java program begins with a class file containing the main method. This class file is converted and all the statically referenced classes in it are queued for processing. The conversion continues from the work queue until all the reachable classes has been converted. The IR is then subjected to many optimizations. Later, register allocation is performed on the optimized IR and finally assembly code is emitted. Marmot does not support dynamic loading [14].

Jalapeno [7], a VM from IBM uses a compile only approach and has three compilers, a Baseline compiler that performs the initial compilation of a method, a quick compiler that does low level of code optimization and an optimizing compiler that compiles frequently executed methods or methods that are computationally intensive. The baseline compiler quickly transforms the bytecode to native code. It imitates the stack machine behaviour of the JVM. Baseline compiler neither creates any intermediate representation nor it does any register allocation. The quick compiler does a low code optimization, primarily register allocation. The optimizing compiler translate the bytecodes into an intermediate representation and performs a series of optimizations, linear scan register allocation and instruction selection. The optimizing compiler produces a high quality code. The optimizing compiler can be invoked as a static compiler or as a dynamic compiler. It is invoked as dynamic to compile the dynamically loaded classes [7]. Jalapeno started as an internal project at IBM and later evolved into a full fledged open source project named Jikes RVM. “Although Jikes RVM often displayed performance which was competitive with production Java virtual machines, it could not run arbitrary Java programs due to unimplemented features in the VM and the libraries" [2].

JET (Just Enought Time) [25] is another AOT compiler which is part of Excelsior JET. Excelsior JET is a mixed compilation Java environment where it uses both an Ahead-Of-Time compiler and a runtime system consisting a JIT compiler. It applies most powerful, time and memory expensive optimizations to achieve an efficient native code. It also applies Java specific optimizations such as method inlining, removal of run time checks, etc. All the classes that were known at compile time gets compiled by the Ahead-Of-Time compiler and the JIT compiler gets invoked when it comes across any dynamic class. The JIT compiler performs weaker optimizations on the dynamically loaded classes due to the time and memory demand of "on the fly" compilation [25].

## 2.5 OptiJava vs. Related Work

OptiJava is an AOT compiler that compiles the Java class files to native executable code. Like all AOT compilers, OptiJava loads the classes and methods and compiles them before run-time. The uniqueness in OptiJava is that it uses coagulation as its compilation technique, which compiles the code at basic block level instead of compiling at method-level. Instruction scheduling and register allocation decisions are done locally for each basic-block. OptiJava also gathers run-time statistics especially the execution frequency of each part of the code. Using this information, it recompiles the class file and produce an efficient native code which could potentially improve the performance of the application.

No previous work has been reported on using code coagulation for object-oriented languages. OptiJava currently has similar limitations to other AOT compilers - except for those that include an interpreter or dynamic compiler - in that it doesn't currently support dynamic loading. Like Caffeine, OptiJava also does not currently

# Chapter 3

# Design Of OptiJava

We have developed an Ahead-Of-Time (AOT) compiler that converts Java class files to native executable code. We attempt to bring the benefits of code coagulation (see Section 2.3.7) to Java compilation. OptiJava restricts its input to verifiable class files. OptiJava loads the classes and methods and does all analysis and code generation at compile time. This will reduce run-time overhead, since at run-time the compiled code can be executed directly. OptiJava compiles `.class` files to native executable code using code coagulation as its compilation technique.

Unlike other traditional AOT compilers, where the compilers compile the bytecode method by method, OptiJava compiles at the basic block level. OptiJava starts by compiling the frequently executed parts of the code known as "hot spots" and gradually expands the compiled code to include the infrequently used parts of the code. OptiJava uses feedback-profiling to get run-time execution information. The OptiJava compiler is entirely implemented in Java, so in the future it will be able to bootstrap by compiling itself. We evaluate the performance of OptiJava using a program that contains frequently executing parts of code.

## 3.1 Structure Of OptiJava

The overall flow diagram of OptiJava is as shown in Figure 3.1. In the initial phase, the classes and methods are loaded. Then, the bytecodes are processed and are converted to Intermediate Representation (IR). In the process of conversion, it creates basic blocks. The basic blocks are connected to form the control-flow graph which is used by the data-flow analysis and code coagulation. The basic blocks are then coagulated using code coagulation, a code generation technique proposed by Karr [21]. At the end of

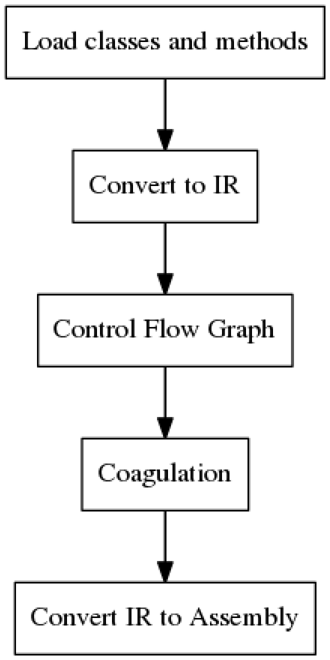


Figure 3.1: Structure of OptiJava

coagulation, instruction ordering and register allocation for each basic block would have been done. The symbolic registers that were allocated are then mapped to the actual physical registers and the IR instructions are translated to architecture specific assembly code in the final phase of OptiJava.

## 3.2 Running example

As a concrete example of the process that OptiJava uses in the compilation process, we will use the example program in Figure 3.2

The `javac` Java compiler produces the java byte code shown in Figure 3.3

## 3.3 Loading of Classes and Methods

Traditional AOT compilers load all the Java methods of a class and compile them. The drawback of this approach is the fact that every single method is loaded, even if its not needed. Since the java libraries contain a lot of methods, loading and compiling all these methods would result in high memory usage. OptiJava only loads those classes and

```
package test;

public class RunningExample {
    static int z = 8;
    public static int code(int p) {
        int s = 1;
        int t = s + 3;
        int a = 20;
        int v = p - 10;
        if (t>p) {
            a = p;
            s = a * 4;
        }
        else
            s=p;
        return p + s;
    }
    public static void main(String []args) {
        int b = 3;
        int c = z + b;
        int y = code(c);
        System.exit(y);
    }
}
```

Figure 3.2: Java source code for running example

```
static int z;
public test.RunningExample();
   0: aload_0
   1: invokespecial #1    // Method java/lang/Object."<init>":()V
   4: return
public static int code(int);
   0: iconst_1
   1: istore_1                    // variable s
   2: iload_1
   3: iconst_3
   4: iadd
   5: istore_2                    // variable t
   6: bipush        20
   8: istore_3                    // variable a
   9: iload_0
  10: bipush        10
  12: isub
  13: istore        4             // variable v
  15: iload_2
  16: iload_0
  17: if_icmple     29
  20: iload_0
  21: istore_3
  22: iload_3
  23: iconst_4
  24: imul
  25: istore_1
  26: goto          31
  29: iload_0
  30: istore_1
  31: iload_0
  32: iload_1
  33: iadd
  34: ireturn
public static void main(java.lang.String[]);
   0: iconst_3
   1: istore_1
   2: getstatic     #2    // Field z:I
   5: iload_1
   6: iadd
   7: istore_2
   8: iload_2
   9: invokestatic  #3    // Method code:(I)I
  12: istore_3
  13: iload_3
  14: invokestatic  #4    // Method java/lang/System.exit:(I)V
  17: return
static {};
   0: bipush        8
   2: putstatic     #2    // Field z:I
   5: return
```

Figure 3.3: Running example JVM instructions

methods that are reachable during some possible run of the program (i.e., OptiJava, will not load methods in the source code that are never referenced). However, an exception to this is while loading virtual methods. While processing a call to a virtual method, the compiler loads methods having the same method signature (method name, number, type and order of the parameters) from all of its superclasses and subclasses of the class type referenced by that instruction.

The three major modules which assist in loading are LoadClass, LoadMethod and LoadBasicBlock.

- LoadClass: responsible for loading the classes and interfaces.
- LoadMethod: responsible for loading the methods.
- LoadBasicBlock: responsible for processing the instructions of the method and thereby creating basic blocks.

These three modules work together in loading the classes and methods. The three modules are interlinked and are mutually dependent on each other.

Initially, the class containing the method main is queued for loading, but before loading this class, it checks if its superclass has been loaded. If the superclass has not been loaded, then, it is queued for loading and its superclass is checked. This recursive checking of the superclasses ends when it comes across loading the superclass which is the root of the class hierarchy, namely, `java.lang.Object`. It then starts loading the classes and interfaces from the root to the bottom of the class hierarchy.

When a class is loaded, its constructors and its class initializer method are loaded by LoadMethod. LoadMethod is responsible for loading all the methods that are referenced in the class. In the process of loading a method, it may call LoadClass to load other referenced classes. It is the responsibility of LoadMethod to identify if the method is a normal (Java) method or a native method. It also identifies if it is a static or a virtual method.

After a method has been loaded, LoadBasicBlock gets invoked on the loaded method. LoadBasicBlock is responsible for processing the instructions of the method and creation of basic blocks. While processing the instructions, it may call LoadMethod on any new method that was referenced in the instruction.

To understand the loading of the classes and methods in OptiJava, consider the example program as shown in Figure 3.2. The steps done in sequence by OptiJava would be as follows:

1. Loading process starts with the class `RunningExample`. Superclasses of this class are loaded by LoadClass. In this case, RunningExample has only one superclass which is `java.lang.Object`. Thus, first `java.lang.Object` is loaded. LoadMethod loads constructors and class initializer method of `java.lang.Object`.

2. Next, LoadClass loads the class `RunningExample`. LoadMethod loads the constructors and the class initializer of the class `RunningExample`. LoadMethod then loads the method `main`. While processing the instructions of `main`, it sees a static call to a method named `code` as well as a static call named `exit`.

3. LoadMethod is invoked to load these new methods. LoadMethod calls LoadClass to load the classes that contain these methods. Method `code` is part of the class `RunningExample`. The method `exit` is part of the library class named `java.lang.System`. Since the class `RunningExample` was already loaded, it proceeds to load the method `code`. The instructions of method `code` is then processed. Next, the class `java.lang.System` is loaded. OptiJava does not load native methods, hence the native method `exit` is not loaded. OptiJava calls native methods directly.

4. Since all the classes and methods that are referenced from the main class have been loaded, the loading phase ends.

Thus, in this manner all the classes and static methods that were needed were loaded. However, while loading a virtual method, OptiJava loads its superclasses and subclasses and also loads all the virtual methods from its superclasses and subclasses that have the same method signature as that of the loaded virtual method. Consider for example the code in Figure 3.4:

The bytecode for the expression `a.add()` in the `main` method is an `INVOKEVIRTUAL` instruction, and the processing of this bytecode sequence forces the loading of all the add methods of its superclasses and subclasses that have the same method signature. Thus, the `add()` method of class A as well as the `add()` methods of classes B and C will be loaded. Note that the `add()` method of class D is not loaded because the method signature of the method `add()` of class A and the method signature of the method `add()` of class D do not match.

Another functionality of LoadMethod is to group all the loaded virtual methods having the same method signature to a method-group. In the example as shown in figure Figure 3.4, the `add()` methods of classes A, B and C would be grouped to a single method-group. Method-groups help in analysing the inter-procedure flow and register

```
package test;

public class A{
    public static void main(String args[]){
        A a = new A();
        a.add();
    }
    void add () {
            System.out.println("In A's add");
    }
}
class B extends A{
    void add (){
        System.out.println("In B's add");
    }
}
class C extends A{
    void add (){
        System.out.println("In C's add");
    }
}
class D extends A{
    void add(int a, int b){
        System.out.println("In D's add");
    }
}
```

Figure 3.4: Java code for a virtual call example

convention between the method that made the vitual call and the virtual methods. All of the methods that could be called from the same place have to have the same register convention.

## 3.4 Conversion to IR

In this phase, the stack based JVM bytecode instructions of the loaded methods are converted to an Intermediate Representation (IR) (see Section 2.3). Within a basic block, OptiJava's IR is a strongly typed, register based, Static Single Assignment (see Section 2.3.2) representation. The IR represents the stack semantics of the original bytecode in terms of a dataflow graph within the basic block. An instruction in the IR consists of an operator which encodes the JVM opcode, some number of operands, and a possible result. The operands are values each of which represents a constant, value passed from another block, array reference, object reference, result of a previous operation, or condition codes.

### 3.4.1 Breaking into Basic Blocks

While processing the bytecode instructions and converting them to IR instructions, a basic block boundary may arise. A basic block is a sequence of instructions that has one entry point and one exit point (see Section 2.3.1). A basic block is created initially at the start of a method. All the instructions get processed in this basic block until we come across a control-flow instruction (unconditional or conditional branch instruction, return, or call instruction), a labelled JVM instruction, or a JVM instruction that can throw an exception. In these situations, a new basic block is created, where the remaining instructions are processed. Additionally, an instruction that can throw an exception will create additional basic blocks to handle the exceptional conditions.

Each basic block has a set of entry values and a set of exit values. Entry values are the values passed in from previous block, with two exceptions:

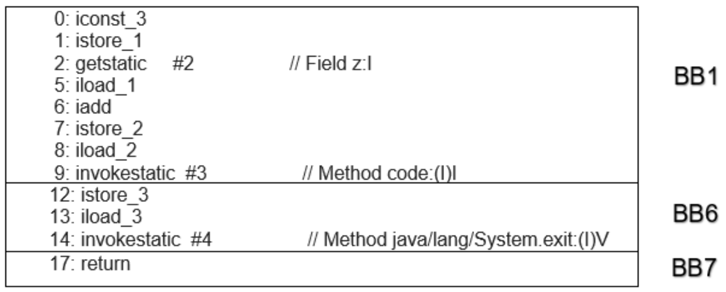
1. the first block of a method: entry values are the local variables plus the parameters;

2. a return point: entry values are the saved values plus the return value.

Exit values are the values that are live at exit from block, that is, their values may be used later, in another block in the control flow.

Each basic block is standalone, data is supplied at its entry and the results are computed according to its IR instructions and these results are provided as its exit values. The values can be thought of as register values and constants, where there is no limit to the number of registers.

Consider the bytecode of the methods `main` and `code` as shown in Figure 3.3. The instructions of method `main` from 0 to 9 are processed in basic block BB1. At instruction 9, since it is a call instruction, a new basic block (BB6) for the called method to return to, is created. The instructions from 12 to 14 are processed in this new block. At instruction 14, since it is a call instruction, a new basic block (BB7) for the called method to return to, is created. The basic blocks that would be created for the method `main` of RunningExample are as shown in Figure 3.5:



```
 0: iconst_3
 1: istore_1
 2: getstatic    #2              // Field z:I
 5: iload_1
 6: iadd
 7: istore_2
 8: iload_2
 9: invokestatic  #3              // Method code:(I)I
12: istore_3
13: iload_3
14: invokestatic  #4              // Method java/lang/System.exit:(I)V
17: return
```

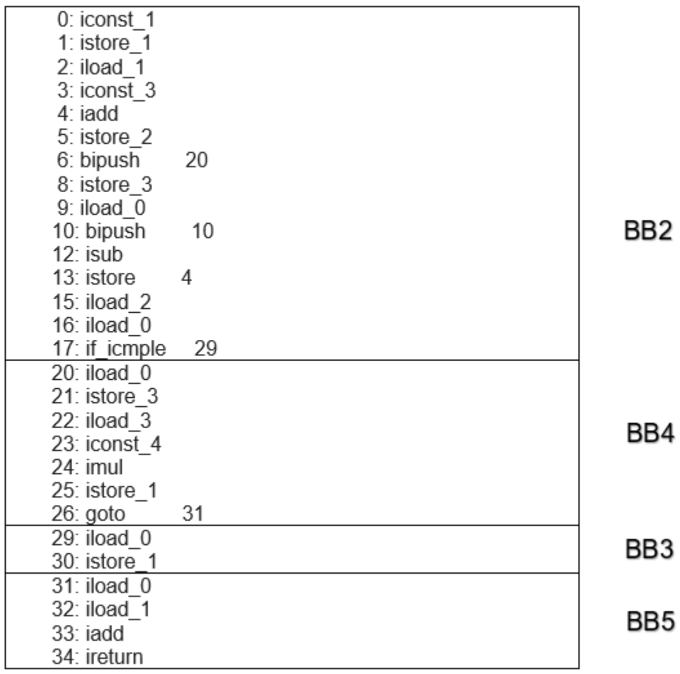
Figure 3.5: Basic blocks of the method main

In similar manner, the bytecode instructions of the method `code` are processed. Instructions from 0 to 17 are processed in the first block. However, when it comes across the compare instruction at bytecode 17, two new basic blocks are created. One basic block for the true case and another basic block for the false case. The processing of the instructions is continued until all blocks have reached the return (or throw) statement of that method. The basic blocks that would be created for the method `code` of RunningExample are as shown in Figure 3.6:

When a new basic block is created, the current state of the stack and local variables become the exit values of the current block, and placeholders are generated for each of these values to become the entry values of the new block. For example, supposing a basic block BB1's instruction resulted in creation of a new basic block BB2, the live values passed from the block BB1 are mapped to values of block BB2.

```
BB1's exit values : r0bb1, r1bb1, r2bb1
```

These are passed as entry values to BB2 and these values get mapped to values of BB2.



```
0: iconst_1
1: istore_1
2: iload_1
3: iconst_3
4: iadd
5: istore_2
6: bipush      20
8: istore_3
9: iload_0
10: bipush      10
12: isub
13: istore      4
15: iload_2
16: iload_0
17: if_icmple    29
```

```
20: iload_0
21: istore_3
22: iload_3
23: iconst_4
24: imul
25: istore_1
26: goto        31
```

```
29: iload_0
30: istore_1
```

```
31: iload_0
32: iload_1
33: iadd
34: ireturn
```

Figure 3.6: Basic blocks of the method code

```
BB2's entry values : r0bb2, r1bb2, r2bb2
```

### 3.4.2 Instruction processing

The bytecode sequence is converted to IR by processing the bytecode instructions sequentially. Processing simulates the JVM's run time stack at compile time. The effects of instruction execution on the stack and local variables are modeled.

The compile time stack maps temporary values of the run time stack to IR values. Type safety in the IR instruction is ensured by mapping the operands of the bytecode instruction to specific IR values, for e.g, if the bytecode operation is an integer operation then the operands are mapped to integer register values and if it is a floating-point operation, the operands are mapped to floating-point register values.

Consider a bytecode instruction "iadd". "iadd" pops the top two elements off the stack, adds them and pushes the result to the stack. The simulated stack before this operation:

```
Stack          : [r0,r1]
```

The simulated stack after "iadd":

```
Stack              :  [r2]
```

The resulting IR instruction would be:

```
IADD r0, r1, r2
```

The operator of the IR instruction is IADD. r0 and r1 are the register values used by the `iadd` instruction which represents the top two integer elements of the stack and the result is in the integer register value r2. Following are the two terms associated with an IR instruction:

- from-values : the value(s) that were used by the instruction; r0 and r1 in the above example.
- to-value : the value that was generated by the instruction; r2 in the above example.

The processing of bytecode sequences results in IR instructions. All the bytecode sequences are processed. However, some bytecode sequences that load or store a constant are not converted to IR instructions.

Consider the bytecode sequences of method main in Figure 3.5. The program counter initially points to the first bytecode instruction. A new basic block (BB1) is created whose entry values are the parameters passed to the method and the local variables used in the method. From the program example as shown in Figure 3.2, it can be understood that the method main has one parameter and three local variables namely b,c and y. Since the local variables are not initialized at this point, they will be represented as null. The initial state would be:

```
Stack              :  []
Local variables    :  [r0bb1, null, null, null]
```

Here, r0bb1 represents the parameter of the method main and nulls are the uninitialized local variables. Local variables are accessed by indexing. For e.g., local variable 0 corresponds to r0bb1 and local variables 1,2 and 3 are nulls. The processing starts from the first bytecode instruction.

0: `iconst_3` When this bytecode is processed, it will push constant 3 to the stack, giving us the modelled stack and local variables:

```
Stack              :  [3]
Local variables    :  [r0bb1, null, null, null]
```

1: `istore_1` When this bytecode is processed, it will pop the top of the stack and stores it into local variable 1, resulting in:

```
Stack           : []
Local variables : [r0bb1,3,null,null]
```

2: `getstatic` When this bytecode is processed, it gets a static field from its corresponding class and the value in the field is pushed to the stack. The static field is resolved by finding the class that has the field and getting the name of the field. This is computed by indexing into constant pool. OptiJava creates a new field constant value (field1) corresponding to the resolved static field and creates a new integer register value(r5bb1) which corresponds to the value of the field. The IR instruction would be to load the int at field1 to rbb15

```
Stack           : [r5bb1]
Local variables : [r0bb1,1,null,null]
IR instruction  : LOAD_INT_AT_FIXED_OFFSET field1 r5bb1
```

3: `iload_1` When this bytecode is processed, it will push the local variable 1 to the stack.

```
Stack           : [3,r5bb1]
Local variables : [r0bb1,3,null,null]
```

4: `iadd` When this bytecode is processed, it will pop the top two elements of the stack and adds them and the result is pushed to the stack. The result gets mapped to an integer register value and this resultant integer register value is pushed to the stack.

```
Stack           : [r6bb1]
Local variables : [r0bb1,3,null,null]
IR instruction  : IADD 3,r5bb1,r6bb1
```

5: `istore_2` When this bytecode is processed, it will pop the top of the stack and stores it into local variable 2.

```
Stack           : []
Local variables : [r0bb1,3,r6bb1,null]
```

6: `iload_2` When this bytecode is processed, it will push the local variable 2 to the stack.

```
Stack           : [r6bb1]
Local variables : [r0bb1,3,r6bb1,null]
```

7: `invokestatic` When this bytecode is processed, it will invoke a static method.

The method to be invoked is resolved by getting name and signature of the method as well as the class in which the method can be found. This information of the method is found by indexing into constant pool.

The elements in the stack are passed as parameters to the called method. If the bytecode instruction is invokevirtual, then an implicit `this` is also pushed to the stack and is sent as the first parameter to the virtual method. The current block of method main ends at invokestatic. An IR instruction, named call-instruction is created in correspondence to the invokestatic bytecode. The exit values of the block will be the locals plus the top of the stack. The top of the stack is the parameter to be passed to the called method.

```
Stack                    : [r6bb1]
Local variables          : [r0bb1,3,r6bb1,null]
Exit Values of block BB1 : r0bb1,3,r6bb1,null,r6bb1
```

A new register value is created for the return of the called method. Then the parameters are popped off the stack and the return register value is pushed to the stack.

```
Stack           : [r7bb1]    \\return register value
Local variables : [r0bb1,3,r6bb1,null]
```

A new basic block is created for the called method to return to. This new block will have its entry as the locals plus the return value.

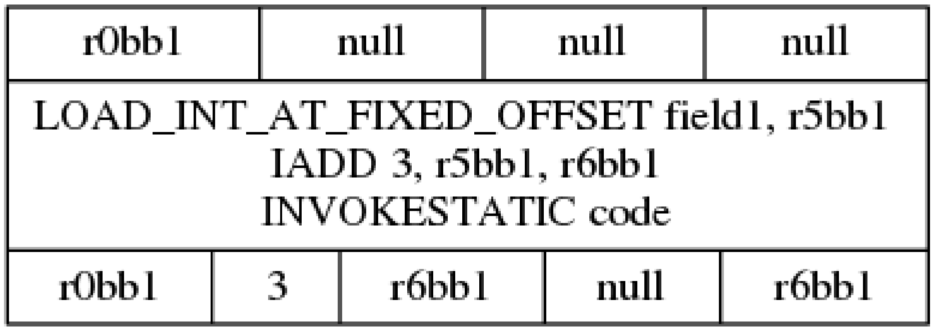
```
Stack           : [r7bb1]
Local variables : [r0bb1,3,r6bb1,null]
Entry Values : r0bb1,3,r6bb1,null,r7bb1
```

The IR instructions and the entry and exit values of the first basic block of the method main is as shown in Figure 3.7. In this way, all the instructions of all the loaded methods are processed and converted to IR instructions.

**Static Single Assignment**

In general, a compiler converts the IR to Static Single Assignment (SSA) form - see Section 2.3.2. However, in OptiJava, SSA is implicit in the IR instructions. This is because the result of a stack operation is always assigned to a new register value. Thus, all the register values will have only one definition for it.

Consider the following code:



| r0bb1 | null | null | null |
|---|---|---|---|

LOAD_INT_AT_FIXED_OFFSET field1, r5bb1
IADD 3, r5bb1, r6bb1
INVOKESTATIC code

| r0bb1 | 3 | r6bb1 | null | r6bb1 |
|---|---|---|---|---|

Figure 3.7: Structure of the first basic block of the method main

```
a = b + c;
d = a /5;
a = d + 10;
e = a * 6;
```

In OptiJava, the IR corresponding to these statements would be:

```
IADD R0, R1, R2        which corresponds to R2 = R0 + R1
IDIV R2, 5, R3         which corresponds to R3 = R2 / 5
IADD R3, 10, R4        which corresponds to R4 = R3 + 10
IMUL R4, 6, R5         which corresponds to R5 = R4 * 6
```

The first definition of varaible 'a' is associated with register value (R2). The redefinition of 'a' is assigned with a new register value (R4) and this new register value is used in the further instructions that uses a.

At the end of processing, all the bytecode instructions are converted to IR instructions that are type safe, register based and SSA form, and each IR instruction is associated with a basic block.

**Basic blocks for native method**

Incase of a native method, a single basic block known as a native basic block is created. This block will be both the first block and the last block of the native method. The entry values of this basic block will be the parameters used by the native method and the exit value will be the return value if any. Unlike other basic blocks whose entry and exit values upon creation are assigned with virtual registers, the entry and exit values of a native basic block will be assigned with the actual physical registers - explained further in Section 2.3.5. Since instructions of a native method are not bytcode, these instructions do not get processed. Instead the native method is called directly.

### 3.4.3 Exception handling

OptiJava implements explicit checks for several run-time exceptions such as null-pointer exception, array index out of bounds exception and arithmetic exception. When an exception check is performed on an instruction, the basic block that contains the instruction will end at that instruction and two new basic blocks will be created. The code which gets executed if there was no exception forms one basic block and another basic block known as exception basic block contains the exception-throwing code. These two basic blocks are then set as targets for the previous block which contained the instruction on which the exception check was performed.

Exceptions within a method is handled by OptiJava. That is, it transfers control to the potential catch block within the method.

*Currently*, if the exception is not within a try-catch block, it is **not** propagated up the call stack to be handled by the caller method. If there is any exception which was not caught by the local catch block then it results in a fatal run time error and execution would be terminated.[1]

## 3.5 Code Coagulation

The basic blocks created from the previous phase are then prepped for coagulation. Coagulation requires the control flow information of the program. Hence, the basic blocks are organized into control flow graph.

### 3.5.1 Control Flow Graph

The basic blocks generated in the previous phase are linked to form the control flow graph. The nodes of the flow graph are the basic blocks and the edges are used to connect the basic blocks. The flow of control can enter a basic block only through the first instruction in the block. Control leaves the block at the last instruction of the block. There is an edge from block A to block B if and only if it is possible for the first instruction in block B to immediately follow the last instruction in block A.

There are certain basic terms that are used by OptiJava to encapsulate the control flow graph. To explain those terms, consider an edge from block A to block B as shown in Figure 3.8. Using this edge as an example, the terms are explained as below:

1. **source :** source is the block of the edge which supplies data. Block A is known as the source of the edge.

[1]See Section 5.1

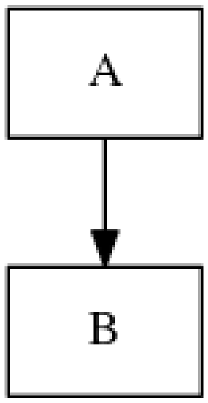


Figure 3.8: Edge between block A to block B

2. **sink :** sink is the block of the edge that receives data. Block B is known as the sink of the edge.
3. **target :** Block B is the target of the block A.
4. **in-edge :** An incoming edge to the block is known as in-edge. Block B is said to have one in-edge. Block A does not have any in-edge.
5. **out-edge :** An outgoing edge from the block is known as out-edge. Block A is said to have one out-edge. Block B does not have any out-edge.
6. **edge-id :** Edge-id is a unique number assigned for each edge. This edge will have its edge-id as 1.

**Intra-method**

Creating edges within a method is straightforward. The basic blocks either ends in a unconditional branch or a conditional branch. In case of a basic block ending in an unconditional branch, an edge is created between the basic block containing the branch instruction and the basic block where it will branch to. If a basic block ends in a conditional branch instruction, then two edges will be created; one edge from the block to the basic block which represents the true case and another edge from the block to the block which represents the false case. In this case, the block is said to have two targets and two out-edges.

**Inter-method**

Creating edges between methods is a little trickier. These edges should represent the control flow among methods. If a block ends in a call instruction, then edges are created which represents the flow of control from the caller method to its callee and the control flow from callee back to its caller. An edge is created between the block that made the call known as caller block and the first block of the called method. We do not create an edge from the called method to its return point. That is, we do not create an edge from the return block of the called method to the caller block's target block. However, this edge is implicit. Finally, an edge is created between the caller block and its target block. The control flow between the methods `main` and `code` of the RunningExample program is as shown in Figure 3.9. The dotted line represents the implicit edge.

If the call instruction was a call to a virtual method, then the called method is connected to all the methods of a method-group which contains those methods whose method signature matches to that of the called virtual method. That is, edges are created between the caller and all of the methods that could be called from the caller. Many methods will be part of a method-group because of overloading. Edges are created from the caller block to the first blocks of all the methods of the matched method-group.

In this manner, edges are built for all the basic blocks that were generated by the previous phase. Figure 3.10 shows the control flow graph of the RunningExample program. In the graph, the basic block NBB1 represents the native basic block generated for the native method `System.exit(y)`.

## 3.5.2 Data-flow Analysis

Data-flow analysis (Section 2.3.2) in OptiJava is both local and inter-procedural analysis. Using the control-flow graph, data-flow analysis (Section 2.3.2) is performed to get the data-flow information. Since, the procedure calls are represented by edges in the control-flow graph, data-flow analysis is implicitly inter-procedural. In addition to this, data-flow within a block is also analysed, this information is used for ordering the instructions within a basic block (explained further in Section 3.6.1).

### Removal of unused values

Sometimes, there may be values which are defined, but, are never used throughout the program. Such values are called unused values. OptiJava does not produce any code for such values. This will reduce the register pressure and will also result in shrinking the size of the native code.

The elimination of unused values is done by traversing the control flow graph in the reverse direction. The process starts from the basic block at the end of the graph that contains the return instruction. If there are any entry values that are not used in the

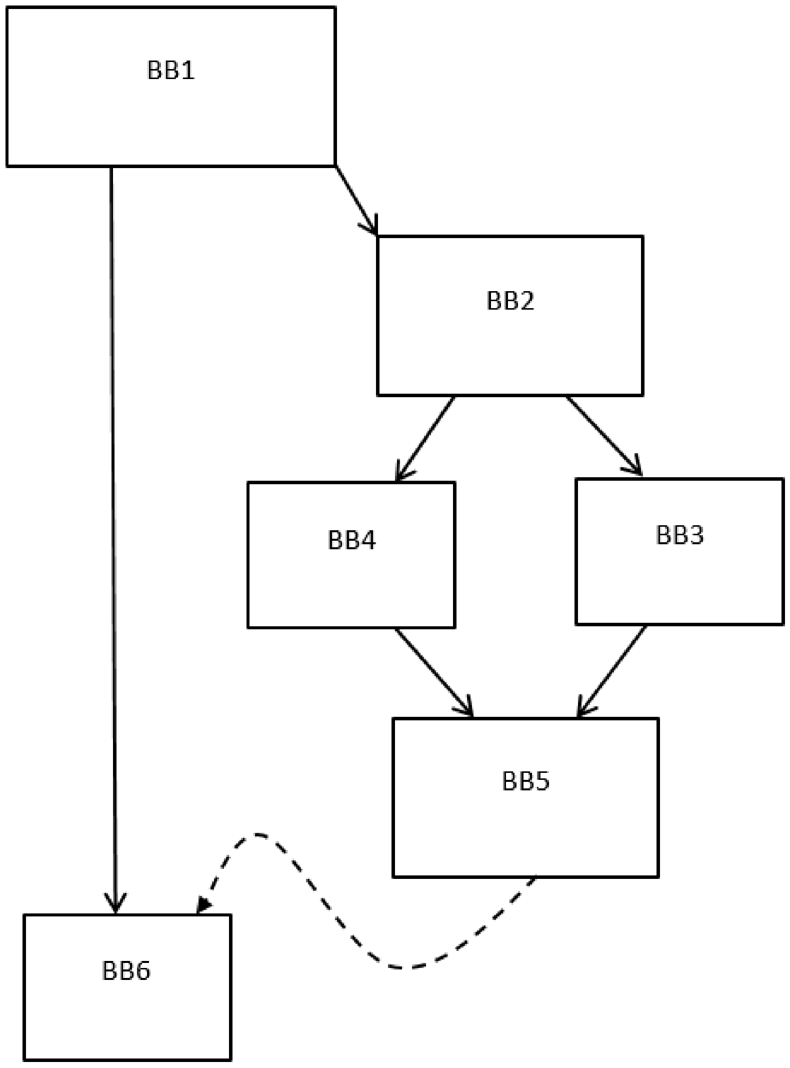


Figure 3.9: Control flow graph between methods main and code

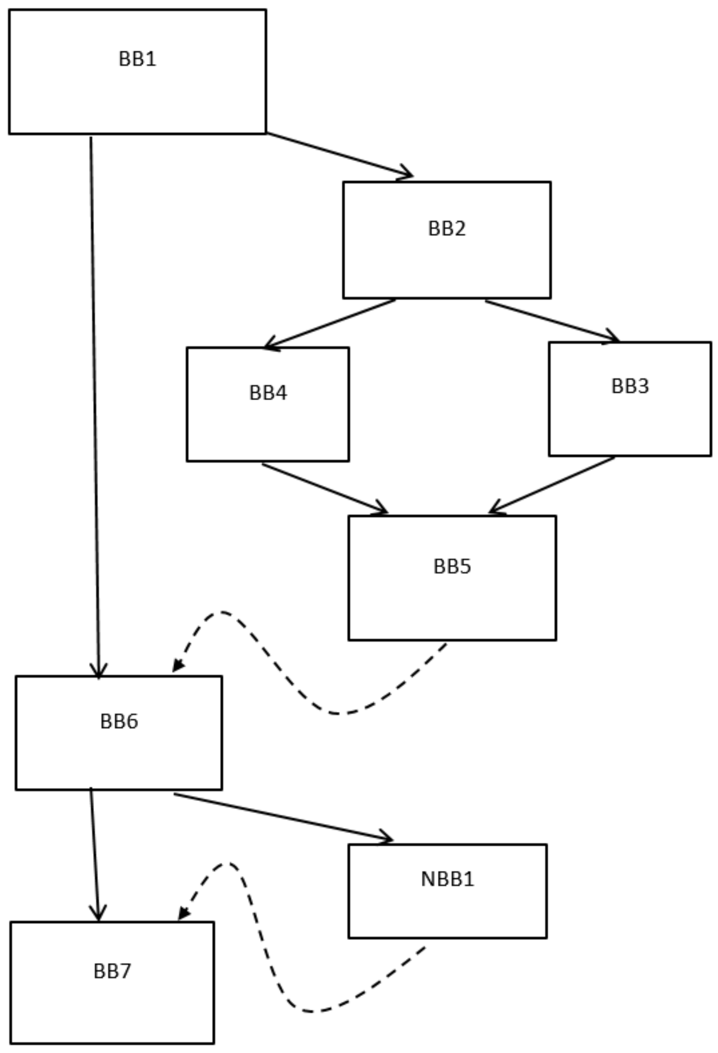


Figure 3.10: Control flow graph of the RunningExample program

instructions of the last basic block, then, that entry value is set to null. The instruction that supplied this entry value is told that it is no longer required in the basic block. And this information is propogated upwards in the control-flow. At the end, the instructions that defined these values gets removed from the instruction sequence.

Consider for example the method `code` of the RunningExample program as shown in Figure 3.2. The variables a and p are not used throughout the program and hence it is safe to remove the instructions that created these values. Bytecode sequences from 6 to 13 in the Figure 3.3 represents the instructions that created these values. Figure 3.11 shows the control flow graph of method `code` of the RunningExample program before removing the unused values. Figure 3.12 shows the control flow of the method code after

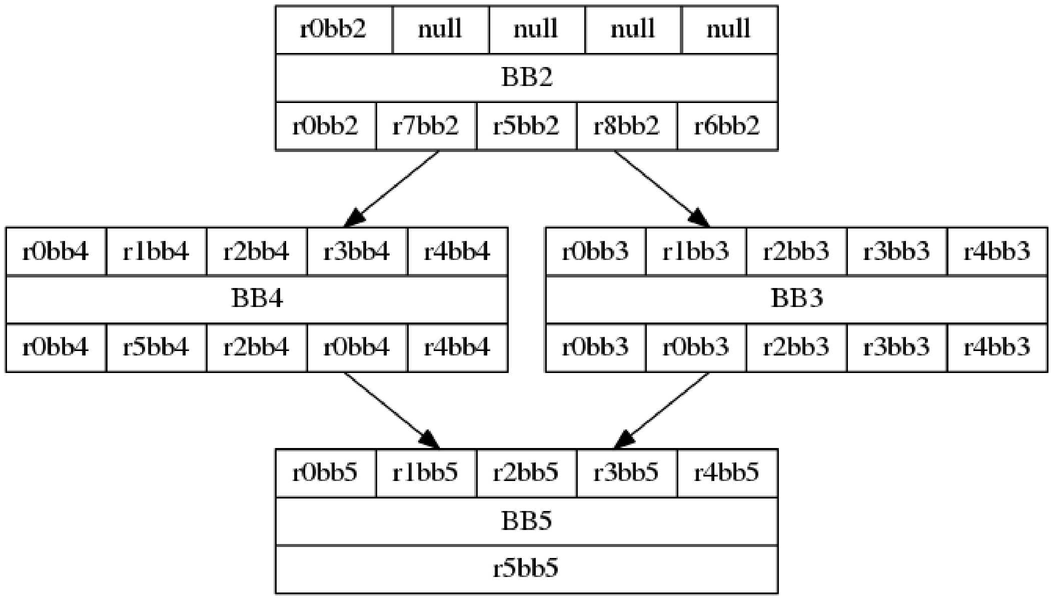


Figure 3.11: Control flow graph of method code

removing the unused values.

### 3.5.3 Frequency Calculation

Optimizations can be more efficient if the compiler knows the execution path at compile time. Profile-directed feedback or Profile-Guided Optimization (PGO) is a two-stage compilation process that provides the compiler with the execution path characteristic of the application's typical behavior after a sample execution. [9] Profiling uses prior annotated runs of the program to generate profile data. A later compilation can then use this profile to guide code optimization decisions in favor of code that executes more

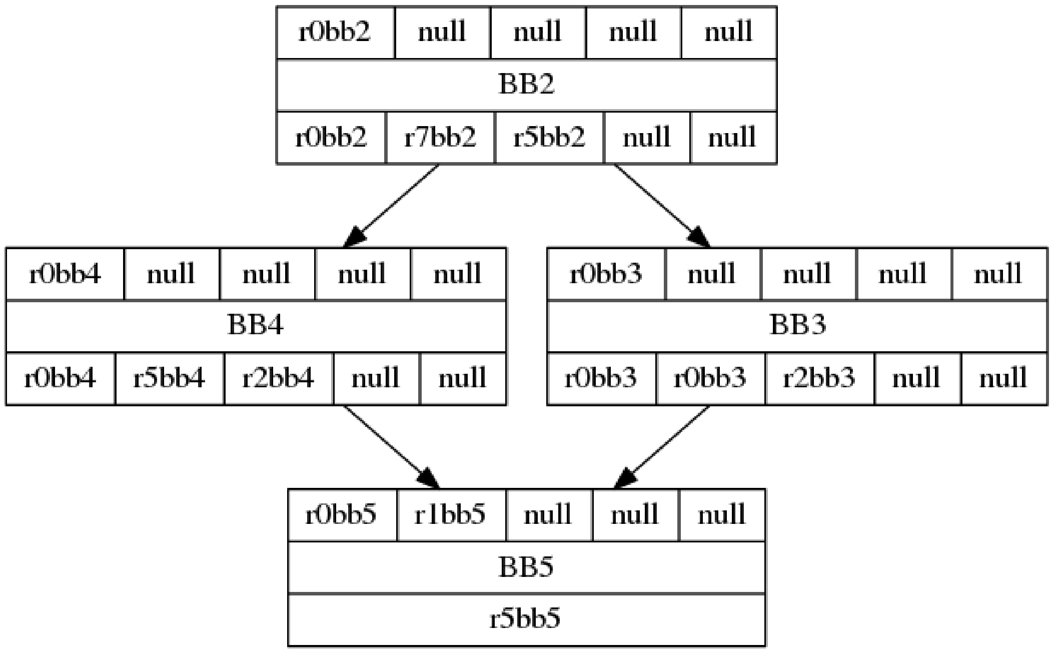


Figure 3.12: Control flow graph of method code after removing unused values

frequently. PGO improves application performance by reorganizing code layout to reduce instruction-cache problems, shrinking code size, and reducing branch mispredictions. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler can be more selective and specific in optimizing the application.

OptiJava collects the execution frequencies (or counts) of the edges as its profile information. The frequency information is used in the coagulation process where the order in which the edges are coagulated depends on its frequency.

OptiJava gathers the execution frequency information of the edges by producing additional native instructions to calculate the execution frequency (count) of the edges. The purpose of these instructions is to record information regarding the control flow of the program. While executing the native code, these extra instructions produce the frequency information. This is done by adding a new intermediate instruction which will increment the frequency count of an edge. An IR instruction to increment the frequency is created for all basic block edges.

The IR instruction to calculate the frequency of an edge is known as “frequency instruction”. Frequency instruction is added to either the source block or the sink block of the edge whose frequency has to be incremented. The frequency instruction will be added to either of the blocks of the edge depending on the following factors:

1. If the edge's source block has just this edge as its out-edge and the edge's sink block has just this edge as its in-edge, then the frequency instruction is added to the source block.

2. If the edge's source block has just this edge as its out-edge and the edge's sink block has many in-edges, then the frequency instruction is added to the source block.

3. If the edge's source block has many out-edges and the edge's sink block has just this edge as its in-edge, then the frequency instruction is added to the sink block.

4. If the edge's source block has many edges as its out-edges and the edge's sink block has many edges as its in-edges, then the following check is done to evaluate the block that can contain the frequency instruction.

   (a) If the source block has a virtual call instruction that has several possible targets due to virtual dispatch, then a load instruction is added to the source block which will load the caller methods offset to a fixed register namely, %r15. The sink block (the called method's first block) will increment the frequency count for the block of sources (because this could have been called from multiple locations), offset by the value passed in %r15.

   (b) Else, a new block is created and inserted in between the source block and the sink block, and the frequency instruction is then added to this new block.

In this manner, the frequency instructions for each edge is added to its respective basic block.

OptiJava uses a default profile information which has the execution frequency of the edges as 1 for its first compilation. At the end of compilation, OptiJava produces a native code that has these frequency instructions. When the native code is run, the frequency instructions also gets executed. At the end of execution of the native code, we get a profile information that contains the number of times each edge was executed. OptiJava uses this new profile information to recompile the class file and produce an efficient native code which could potentially improve the performance of the application.

### 3.5.4 Coagulation

Coagulation is the core of OptiJava. The main idea behind coagulation is to optimize and compile small regions of code locally in isolation and then merge the compiled parts of the program. The basic blocks generated by the previous phase are small regions of the program. These blocks are solidified, and the solidified blocks are merged together (see Section 2.3.7).

Solidification of a block is a process where the instructions of the block are placed in an efficient order of execution and the operands of the register values which would to this point have been virtual registers are mapped to generic/fixed registers (explained in detail in Section 3.6). This is different from traditional compilers, where instruction selection, instruction scheduling and register allocation is done on method level. Conventional compilers use heuristics of the edge frequency to determine the program's structure in advance and perform optimizations based on these heuristics. Coagulation technique has more information about the runtime behaviour of the code and can perform more optimal local optimizations.

Following are the steps of the coagulation phase:

1. All the edges are queued for coagulation.
2. The highest (remaining) frequency edge is removed from the queue and used for coagulation.
3. If the edge has no solidified blocks at either end then one of the blocks must be solidified first. The sink of the edge is solidified first, unless the source has many targets or the source ends in a call.
4. The exit values of the source block and the entry values of the sink block of the edge are unified. If these values can be unified the blocks are merged. If both blocks were already solidified this usually isn't possible, so a minimal cost repair such as inserting a copy instruction is done to align the values.
5. The other uncompiled block of the edge is solidified if it wasn't already.
6. Steps 2 to 5 are repeated until all the edges in the queues are coagulated.

### 3.5.5 Unification

Unification is a common compiler technique. The goal of unification of two terms is to find a most general substitution of the variables occuring within the terms such that the two terms become equal.[1]. Unification is done as part of coagulation where the exit values of the source block and the entry values of the sink block of an edge are unified. We use "union-find algorithm" to unify the values.[1].

Union-find algorithm involves two operations: `union()` and `find()`. `union()` merges two values and makes one of the values as the representative of the two values. `find()` returns the representative value.

To explain unification in detail, a few specific terms used in OptiJava are useful to understand:

As an implementation detail, "operands" are assigned to the values. `Operands` represents the registers or constants that are assigned to the values. A constant value will be assigned with a constant operand that represents the actual constant. A register value will be assigned with a register operand. The register operands are of four types:

- Virtual registers
  During creation of an IR instruction, compiler ignores the fact that the target machine has a limited set of architectures, instead, it assigns an unlimited set of virtual registers to the register values used in the IR instructions and assumes that enough registers existed. These registers are not related to any physical storage location and therefore they are merely tags.

- Generic registers
  Generic registers are fixed number of virtual registers. This number depends on the number of general purpose registers of the underlying architecture. A x86-64 architecture has 16 general purpose registers and hence the generic registers will be 16 for a x86-64 architecture. The virtual registers are mapped to generic registers by register allocator.

- Spill registers
  Spill registers represent the memory location of the value. During register allocation, there may be scenarios where a register value was not assigned with a generic register because there were not enough registers. In such a scenario, the register value is stored in memory.

- Fixed registers
  Fixed registers are the actual physical registers/general purpose registers of the underlying architecture. The generic registers are converted to fixed registers in assembly phase.

The goal of coagulation is to have code flow as seamlessly as possible. Unification makes this happen. Basic blocks have unassigned registers. The registers output from a basic block must allign with the registers input to a subsequent basic block. Unification tries to unify the register output of a block to register input of another block. This would result in continous flow of data in the same registers thereby reducing register pressure. If the register values cannot be unified then they have to be adjusted. By design, these adjustments happen at the less frequently executed parts of the code. The actual hardware register assignment happens later, see Section 3.7.

To perform `union()` of two values, we first evaluate `find()` of each of the values and then check to see if the operands that were assigned to `find()` of these values satisy the unification rules. The unification rules are:

1. **Unification between two constants:** Unification succeeds if the two constants are equal and fails otherwise.

2. **Unification between a constant and a virtual register:** Unification succeeds.

3. **Unification between a constant and a generic/fixed register:** Unification fails.

4. **Unification between two virtual registers:** Unification succeeds irrespective of whether the virtual registers are equal or different.

5. **Unification between a virtual register and a generic register:** Unification succeeds.

6. **Unification between a virtual register and a fixed register:** Unification succeeds.

7. **Unification between a virtual register and a spill register:** Unification succeeds

8. **Unification between two generic registers:** Unification succeeds if the two generic registers are equal and fails otherwise.

9. **Unification between a generic register and a fixed register:** Unification succeeds on two conditions:

    - If the generic register was mapped to the same fixed register.
    - If the generic register is previously not mapped to any fixed registers and the fixed register was not mapped to any other generic registers.

    Unification fails otherwise.

10. **Unification between two fixed registers:** Unification succeeds if the two fixed registers are equal and fails otherwise.

11. **Unification between a generic/fixed register and a spill register:** Unification fails.

If the unification succeeds based on the above rules, then the `union()` on the values are performed and the `find()` will now return the new representative value of the unification. On successfull unification, the values and constants gets propagated down the control flow.

If the unification fails, then the values cannot be merged. The exit value of the source will be copied/spilled/unspilled to the entry value of the sink value.

- **Copy:** If the operand of either the exit value of source or the entry value of sink is not a spill register and the unification rules were not satisfied then the soure's exit value is copied to the sink's entry value.

- **Spill:** If the operand of the exit value of source is a generic/fixed register and the operand of the entry value of sink is a spill register, then the source's exit value is spilled to the sink's entry value.

- **UnSpill:** If the operand of the exit value of source is a spill register and the operand of the entry value of the sink is a generic/fixed register, then the source's exit value is unspilled to the sink's entry value.

Apart from the operand rules, the source value will be copied to the sink value based on the following conditions:

1. If the exit values of the source contains duplicate values (more than one occurance of the same value).

2. If the operand of any of the exit values matches to any of the operands of entry values of the sink.

The copy, spill and unspill from a source's exit value to a sink's entry value is done by creating a new IR instruction known as copy instruction, spill instruction and unspill instruction respectively. These instructions are either added to the source block or the sink block depending on the following factors:

1. **Solidified source block and unsolidified sink block:** In this case, since the instruction ordering and the register convention of the source block has already been decided, the copy, spill and unspill instructions cannot be added to the source, hence these instructions will be added to the sink block.

2. **Solidified sink block and unsolidified source block:** In this case, since the instruction ordering and the register convention of the sink block has already been decided, the copy, spill and unspill instructions cannot be added to the sink, hence these instructions will be added to the source block.

3. **Both source and sink are solidified:** In this case, since the instruction ordering and the register convention of both the blocks has already been decided, the copy, spill and instructions cannot be added to either of the blocks. In this case, a new basic block known as copy block is created between the source block and the sink block.

### 3.5.6 Coagulation with example

The entire process of coagulation is explained below using the edges from the control flow graph of the RunningExample program as shown in Figure 3.10. In the initial compilation of the program by OptiJava, it considers all the edges to have the same frequencies. Hence, the edges from the queue will be taken in random order for coagulation. Supposing the edges taken from the queue were in order of :

- Edge BB2→BB4
- Edge BB2→BB3
- Edge BB4→BB5
- Edge BB3→BB5
- Edge BB1→BB2
- Edge BB1→BB6
- Edge BB6→NBB1
- Edge BB6→BB7

**First, the edge BB2→BB4 is coagulated**.

- The edge BB2→BB4 has both its source block and sink block as unsolidified blocks.
- Block BB2 is solidified first because BB2 has two outEdges.

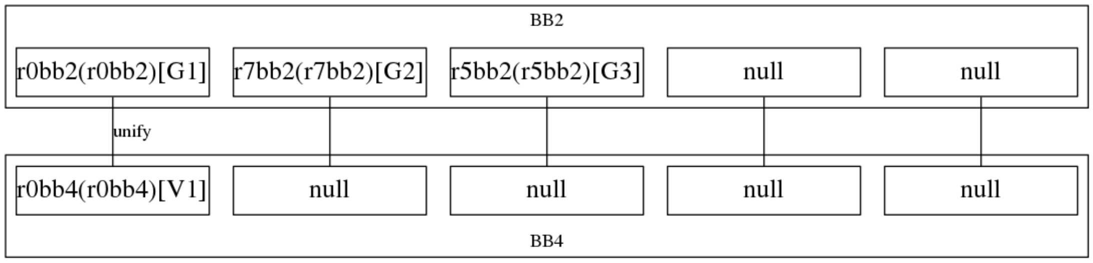


Figure 3.13: Unification of exit values of block BB2 and entry values of block BB4

- The exit values of BB2 are then unified with the entry values of BB4. The exit values of the block BB2 and the entry values of the block BB4 are as shown in Figure 3.13. The `find()` of the values are shown in parenthesis. `Operands` assigned to the `find()` of the values are shown in square brackets.

  Unification of source's exit values and sink's entry values is performed in the following way:

  – At position 0, since the operand of the source's entry value is a generic register (G1) and the operand of sink's entry value is a virtual register (V1), unification succeeds, hence, the `union()` of values r0bb2 and r0bb4 is performed. The `find()` of r0bb4 is now r0bb2.

  – At positions 1 to 2, since the entry values of the block BB4 are null, unification is not performed.

  – At positions 3 to 4, since both the entry values and exit values are null, unification is not performed. The state of the values after unification is as shown in Figure 3.14.

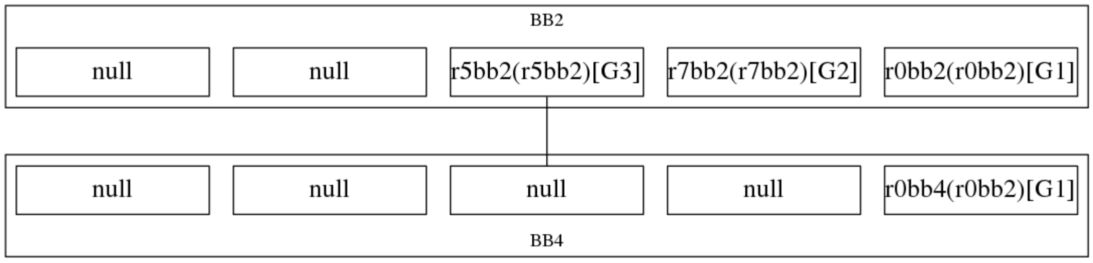


Figure 3.14: The exit values of block BB2 and entry values of block BB4 after unification

- After unification, the block BB4 is solidified.

**Next edge from the queue is BB2→BB3**.

- The edge BB2→BB3 has both its source block BB2 as solidified and sink block BB3 as unsolidified block.
- The exit values of BB2 are unified with the entry values of BB3. The state of the values after unification is as shown in Figure 3.15

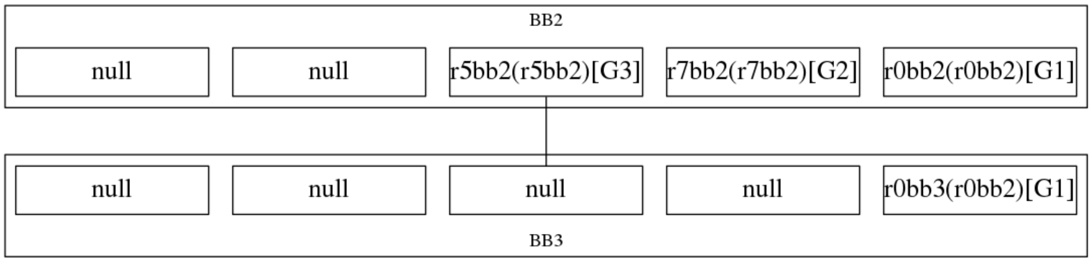


Figure 3.15: The exit values of block BB2 and entry values of block BB3 after unification

- After unification, the block BB3 is solidified.

**Next edge from the queue is BB4→BB5**.

- The edge BB4→BB5 has its source block BB4 as solidified and sink block BB5 as unsolidified block.
- The exit values of BB4 are unified with the entry values of BB5. The state of the values after unification is as shown in Figure 3.16

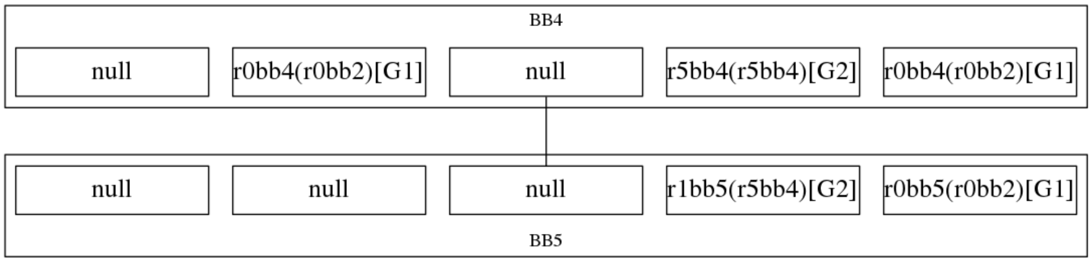


Figure 3.16: The exit values of block BB4 and entry values of block BB5 after unification

- After unification, the block BB5 is solidified.

**Next edge from the queue is BB3→BB5**.

- The edge BB3→BB5 has both the source block BB3 and sink block BB5 as solidified blocks.
- The exit values of BB3 are unified with the entry values of BB5. The state of the values after unification is as shown in Figure 3.17 Since the exit value r0bb3

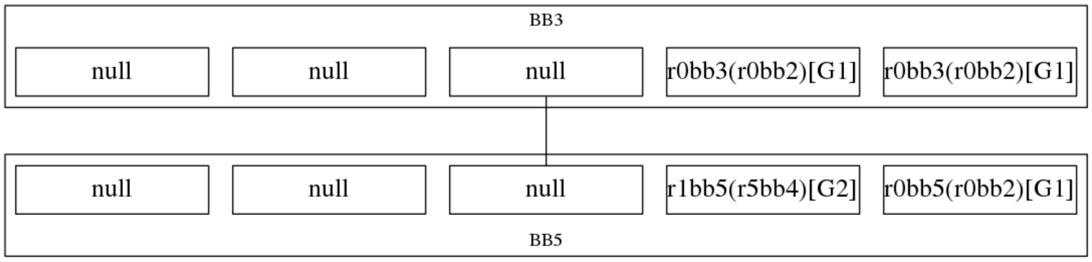


Figure 3.17: The exit values of block BB3 and entry values of block BB5 after unification

occured both at position 0 and at position 1, a copy instruction had to be created from the exit value r0bb3 at position 1 to the entry value r1bb5 at position 1. But since both the blocks are solidified, the copy instruction cannot be added to BB3 or BB5, hence, a new block known as copy block (CBB1) will be created. The copy instruction is then added to this copy block. The target of BB3 is changed from BB5 to CBB1. And target of CBB1 will be BB5.

**Next edge from the queue is BB1→BB2**

Coagulation of this edge is slightly different from the other edges due to the fact that the basic block BB1 ends in a call instruction `INVOKESTATIC`. BB1 belongs to the method main which makes the call to the method code. BB2 belongs to the method code which was being called. This edge is a call-edge , i.e. this is the edge between a block that ends in a call instruction, and the called methods first block.

To preserve the inter-procedure flow information among the methods main and code, three edges are considered together for coagulation; the call-edge, the implicit return edge, and the caller's target edge. The return-edge is an implicit edge between the called methods return block and the target of the caller block. The target-edge is the edge between the caller block and its target.

The control flow graph among the method main and code along is as shown in Figure 3.18. The edges of importance are:

- Call-edge : BB1→BB2
- Return-edge : BB5→BB6

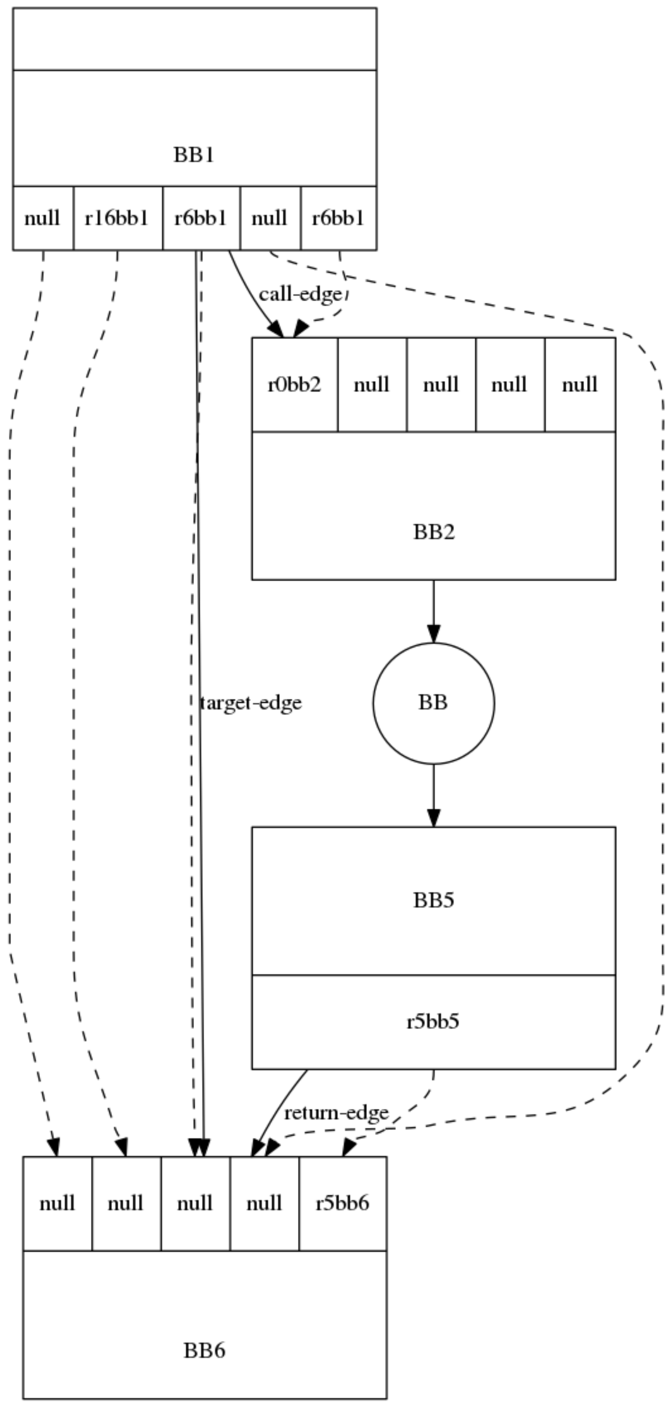


Figure 3.18: Control flow graph representing the control flow among methods main and code.

- Target-edge : BB1→BB6

The order of coagulation among these edges would be : first the edge BB1→BB2 followed by the edge BB5→BB6 and finally the edge BB1→BB6. Unification of these blocks differs slightly.

1. **BB1→BB2 :** While unifying the blocks BB1 and BB2, the parameters that are passed from BB1 to the block BB2 is unified with the parameters of block BB2.As shown in Figure 3.18, the parameter passed from BB1 is r6bb1 and the parameter of the method code is the first entry value of the block BB2. Thus, r6bb1 is unified with r0bb2.

2. **BB5→BB6 :** While unifying the blocks BB5 with BB6, the exit value of BB5, which is the return value of the method code, is unified with the return point of block BB6.As shown in Figure 3.18, the return value of the method code is the exit value from the block BB5 and the return point is the last entry value of the block BB6. Thus, r5bb5 is unified with r5bb6.

3. **BB1→BB6 :** While unifying the blocks BB1 with BB6, the live values from BB1 are unified with the corresponding entry values of the block BB6.

After these edges has been coagulated, the next edge from the queue is taken for coagulation. In this manner, all the edges are coagulated in the order in which they were taken from the queue.

When the class file is compiled for the second time, it has the actual execution frequency information of the edges and hence the edges will be taken from the queue according to their priority. The highest frequency edge is given higher priority for coagulation. Thus, the order of coagulation will start from the highest frequency edge to the lowest frequency edge. Supposing after the program was run, if the code in the true case of the conditional was executed, then the frequency of the edge BB2→BB4 would have the highest frequency, followed by the edge BB4→BB5. The new coagulation order would then be:

- Edge BB2→BB4
- Edge BB4→BB5
- Edge BB2→BB3
- Edge BB3→BB5

- Edge BB1→BB2
- Edge BB1→BB6
- Edge BB6→NBB1
- Edge BB6→BB7

Coagulation starts by compiling the most frequently executed parts of a program, for example, loops and recursive calls, and locally allocates optimal resources to them. When two already-compiled parts of the program are merged, adjustments may have to be made to ensure the availability of resources previously assumed to be available. However, these adjustments typically occur at less frequently executed parts of the program.[26]

At the end of coagulation phase, each basic block has been solidified, which means that register allocation and instruction selection has been done. All that remains is to assign physical registers to the symbolic registers and translate the IR instructions to assembly - all of which is architecture specific.

## 3.6 Solidification

Solidification is the process of ordering the instructions and allocating registers for the register values used in those instructions. Solidification is done locally, that is, it is done on each basic block. The IR instructions of the basic block are ordered in such a manner that it preserves the semantics of the basic block as well as addreses the dependencies between each instruction. Once the instructions have been ordered, the basic block is then passed on to the register allocation phase where the entry register values, exit register values of the block, and the register values used in the IR instructions of the block are allocated with generic registers. The register allocation used by OptiJava is a graph coloring algorithm. The goal of instruction ordering and register allocation is to reduce the demand of the registers.

"At any point a solidified block, is essentially a self-contained program in the sense that, data was supplied at its entry according to its demanded boundary conditions, it would compute according to its intermediate code semantics and provide results on its exit consistent with its supply boundary conditions."[26]

### 3.6.1 Order Instructions

Ordering the instructions is the arrangement of instructions of a basic block in an efficient order while ensuring that the semantics of the basic block are maintained. While ordering the instructions, data dependency and side effects among the instructions have to be considered. An instruction j is said to be data dependent on instruction i if instruction i produces a result that may be used by instruction j. In this case, instruction i has to be exceuted before instruction j [17]. Since data-flow in IR instructions occurs in registers, detecting the dependencies is straightforward. Additionally there are instructions that are side effects. A side effect represents an instruction that either does not depend on result from a previous instruction or does not produce a result for a subsequent instruction. Rather, a side effect depends on a value in memory or stores a value in memory. The side-effect instructions must maintain their source-code relative order while fitting in with the dependencies of the other instructions.

Within a basic block, the data-flow analysis to analyse the dependency among the instructions can be evaluated by creating a data dependency graph [16]. The dependency graph is created by tracing all the instructions that resulted in producing the corresponding exit value.

Consider the basic block BB2 of method code as shown in Figure 3.12. The exit values and the side effect value of BB2 are as shown below:

```
Exit values         : r0bb2, r20bb2, r5bb2, null, null
Side effect value   : sebb2 (side effect of the compare instruction)
```

The dependency is evaluated by tracing in reverse order of all the instructions that resulted in producing each exit value and side effect.

1. Evaluate the instruction that produced the exit value.
2. Get the from values of this instructions or in other words, get the values that are used by this instruction.
3. Evaluate the instruction that produced the value got in step 2.
4. Repeat steps 2 and 3 until all the values have been traversed.

Figure 3.19 shows the Directed Acyclic Graph(DAG) representation of the data dependency between the instructions of the block BB2. Each circular node represents the IR instruction, rectangular nodes represents the values used by the instruction and each arc represents the dependency between the values. Figure 3.20 shows the Directed Acyclic Graph(DAG) representation of the data dependency between the instructions that resulted in the side effect value.

Consider the exit value r5bb2 of block BB2. From the DAG representation shown in Figure 3.19, it can be deduced that the instructions that resulted in producing the value r5bb2 are in the order :

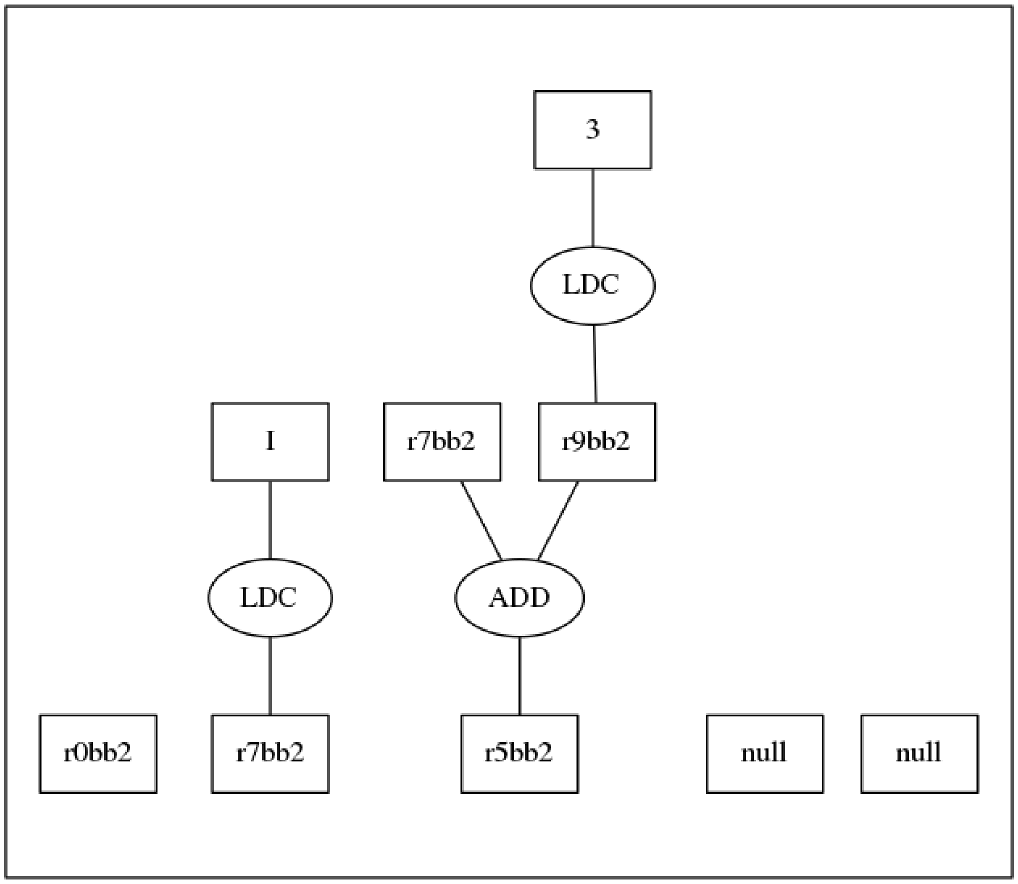


Figure 3.19: DAG representing the operations of the instructions of the block BB2

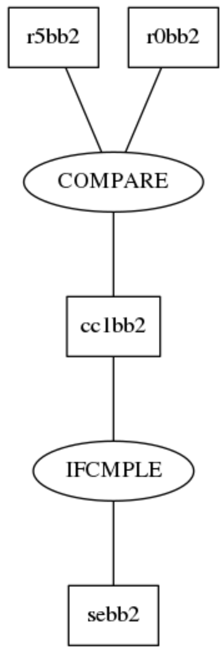


Figure 3.20: DAG representing the operations of the instructions of the side effect in BB2

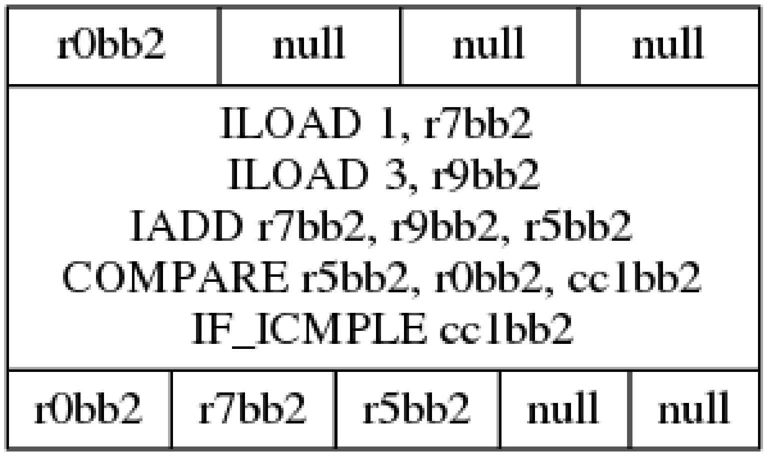
```
ILOAD  1,r7bb2
ILOAD  3,r9bb2
IADD   r7bb2,r9bb2,  r5bb2
```

In this manner the instructions within the block are ordered by calculating the dependency of all the exit values and the side effects of the block. The instructions of the block BB2 in order is as shown in Figure 3.21. Once the instructions have been ordered, register allocation is performed for this block.

### 3.6.2 Register Allocation

Register allocation is the phase where the virtual registers are mapped to a finite number of machine (or, physical) registers while taking care to maintain the semantic of the program (see Section 2.3.5).

Graph coloring does register allocation by building an interference graph that models when two live ranges cannot reside in the same register. It then heuristically attempts to color the graph with the number of colors being equal to the number of physical registers available in the target architecture.. If the heuristic succeeds, the coloring results in a register assignment. If it fails, some register candidates are spilled to memory, spill code



```
r0bb2 | null | null | null
ILOAD 1, r7bb2
ILOAD 3, r9bb2
IADD r7bb2, r9bb2, r5bb2
COMPARE r5bb2, r0bb2, cc1bb2
IF_ICMPLE cc1bb2
r0bb2 | r7bb2 | r5bb2 | null | null
```

Figure 3.21: IR instructions of the block BB2 after Instruction Ordering

is inserted for their occurrences, and the whole process repeats.

Chaitin et M. [8] first used graph coloring as a paradigm for register allocation and assignment in a compiler. Chaitin's allocator used to spill the chosen value everywhere. That is, it placed a STORE instruction after each definition of the value and a LOAD instruction before each use of the value. Over the years many modifications to this basic algorithm has been proposed. One such modification was to reduce the spill code by live range splitting.

We use a register allocation algorithm that is similar to Cooper's live range splitting algorithm.[10] However, instead of global allocation, register allocation in OptiJava is done locally. That is, the allocation is done within each basic block. Register allocation in OptiJava maps the virtual registers to generic registers instead of mapping to physical registers. The mapping of generic registers to fixed registers is delayed as much as possible to give a greater flexibility for register allocation, hence, it is done in assembly phase.

The flow of register allocation algorithm is as follows:

**Liveness analysis**

Liveness analysis calculates the live range of all the register values used in the basic block including the entry and exit register values of the block. Live range is generated by calculating the definition and the last use of each value. Once the liveness analysis is complete, this information is used to build the interference graph.

**Build**

Interference graph is then built using the liveness analysis information. The interference graph contains a node for each register value and an edge between each pair of register values that are simultaneously live. That is, an edge exists between two register values if the live range of the register values interfere with each other.

**Color**

Coloring is a two step process. During the first phase, we repeatedly try to assign a color to a node different from those of its neighbors. If no color is available, a register value is chosen whose live range will be splitted. If we are able to assign a color to every live range, this corresponds to a valid allocation, and the algorithm terminates.

If the coloring is not a success, then a register value is chosen for spilling based on heuristics. Then the live range of the chosen register value is split into smaller ranges. A new register value is created which corresponds to the splitted live range. The code is updated to keep the chosen register value in memory. The instructions are traversed, inserting LOADs and STOREs for the chosen register value.

Next, liveness analysis is again performed and the interference graph is re-built using the new liveness information and the register values are colored. If the coloring fails, the entire process is repeated until all the register values gets assigned with a color.

Consider the block BB2 as show in Figure 3.21. The IR instructions of block BB2 are:

```
0: ILOAD 1, r7bb2
1: ILOAD 3, r9bb2
2: IADD r7bb2, r9bb2, r5bb2
3: COMPARE r5bb2, r0bb2, cc1bb2
4: IF_ICMPLE cc1bb2
```

The register values used in this block are:

```
r0bb2
r7bb2
r9bb2
r5bb2
cc1bb2
```

Live range is calculated for each of these register values. The start range of entry values of the block is marked as -1. The start range of other register values is the instruction number where the value got defined. The end range of exit values of the block is the total number of instructions plus 1. The end range of other register values is the instruction number where it was used last.

Following is the live range for the register values used in block BB2.

```
r0bb2   = [-1,5]
r7bb2 = [0 ,5]
r9bb2 = [1 ,2]
r5bb2   = [2 ,5]
cc1bb2 = [3 ,4]
```

The interference graph is then built using this live range information. The interfernce graph of block BB2 is as shown in Figure 3.22. The nodes of the graph represent all

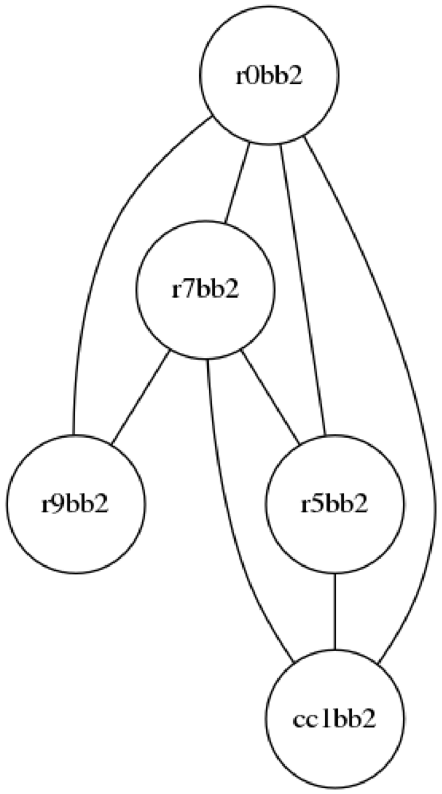


Figure 3.22: Interference graph of register values used in block BB2

the register values used in the block and the edges represent the conflicts between them. That is, if there is an edge between two register values it indicates that the register values are live at the same time and hence cannot be allocated with the same generic register.

Next, it tries to allocate color for each of the register value. Allocator tries to allocate as few colors as possible, but the nodes which are connected cannot be allocated with the same color. Consider the number of colors available were 5.

1. `r0bb2`
   Initially, color 1 is allocated for this register value.

2. `r7bb2`
   While allocating color for r7bb2, the same color cannot be allocated since it conflicts with r0bb2. Hence, r7bb2 gets a new color, namely, color 2.

3. `r9bb2`
   r9bb2 conflicts with both r0bb2 and r7bb2 and hence r9bb2 gets a new color which is different from both r0bb2 and r7bb2. Thus, color 3 is allocated for this register value.

4. `r5bb2`
   r5bb2 conflicts with both r0bb2 and r7bb2 but does not conflict with r9bb2. Hence, a color which is different from r0bb2 and r20bb2 but can be similar to r9bb2 is chosen. Thus, color 3 is allocated for this register value.

5. `cc1bb2`
   cc1bb2 conflicts with r0bb2,r7bb2 and r5bb2. Hence, a color which is different from these register values has to be chosen. Thus, color 4 is allocated for this register value.

This resulted in a successful coloring.

However, consider a scenario where there were not enough colors available. In such a scenario, a register value will be chosen for live range splitting. Then, the live range is split and spill code is inserted. The instruction for storing a register value to memory is known as spilling, an IR instruction known as spill instruction is created to represent this operation. The instruction for loading the value from memory to register value is known as unspilling, an IR instruction known as unspill instruction is created to represent this operation.

Consider a hypothetical situation where the number of available colors are 3. Consider for example, the same block BB2 which was used above. Since four colors are required for a successful allocation and there are only three colors available, the register allocation for the block BB2 will now fail. Hence, a value is chosen for spilling. Consider r7bb2 was chosen for spilling. The live range of r7bb2 is split into two smaller live ranges. Spill code is then added. The new instructions after inserting the spill code would be as shown below:

```
0: ILOAD 1, r7bb2
```

```
1: SPILL r7bb2, spill1
2: ILOAD 3, r9bb2
3: UNSPILL spill1 , r7bb2‘
4: IADD r7bb2‘, r9bb2, r5bb2
5: COMPARE r5bb2, r0bb2, cc1bb2
6: IF_ICMPLE cc1bb2
```

Note that the live range of r7bb2 is now from instruction 0 to instruction 1. The new live range of r7bb2‘ is from instruction 3 to 4. Interference graph is then rebuilt using the new liveness information. Next, the graph is attempted to color. If it succeeds, the algorithm is terminated , else the process continues until the graph can be colored with the available number of colors.

The number of colors available for allocation depends on the actual number of physical registers available in the architecture and reserved registers used by OptiJava. OptiJava reserves two registers; one for the stack pointer and another for frequency calculation. These registers are not used for register allocation. In the initial compilation of the class files, OptiJava reserves a register for frequency calculation and once it has the frequency information the reserved register is marked free and can be used for allocation when the class files are compiled for the second time. In x86-64 architecture, the number of physical registers are 16, OptiJava reserves two registers, so the total number of colors available for allocation will be 14.

Once all the register values have been allocated with a color, the actual register assignment takes place, where, each register value is assigned with a generic register. This is done by mapping the colors with unique generic registers. The register values after register allocation is as shown below; The generic register assigned for each of the register values is shown in square brackets.

- r0bb2←[G1]
- r7bb2←[G2]
- r9bb2←[G3]
- r5bb2←[G3]
- cc1bb2←[G4]

**Pre-colored nodes**

There may be scenarios where the nodes were already assigned with generic/fixed registers. In such a scenario, the colors of the nodes will be mapped to the assigned generic registers instead of mapping to new generic registers.

**Calling convention**

Calling convention is a set of rules that determine how a method will recieve its parameters and how it will pass the result. It defines the Application Binary Interface (ABI) for the program control flow to transfer into and out of a method, that is, how to pass arguments and return values, and how to save and restore the registers that are used across calls. In a method call, we refer to the method that is invoked as the callee and the calling method as the caller. Calling convention also defines how the registers are preserved across calls. For this reason, it divides the registers into caller-saved registers and callee saved registers. [3]

**Caller-save registers**

The registers designated for the caller to save are caller-saves registers. In general, the caller-save registers are used to hold temporary quantities that need not be preserved across calls. For that reason, it is the caller's responsibility to push these registers onto the stack if it wants to restore this value after a procedure call.[3]

**Callee-save registers**

The registers designated for the callee to save are callee-save registers. In general, the callee save registers are used to store values that should be preserved across calls. When the caller makes a procedure call, it can expect that those registers will hold the same value after the callee returns, making it the responsibility of the callee to save them and restore them before returning to the caller.[3]

According to x86-64 ABI [24], the first 6 arguments are passed via registers and the rest are passed on stack. The integer arguments are passed in order:

```
%rdi, %rsi, %rdx, %rcx, %r8, %r9
```

The result is stored in %rax register.
The designated caller-save registers are:

```
%rax, %rcx, %rdx, %rsi, %rdi, %r8, %r9, %r10, %r11.
```

The designated callee-save registers are:

```
%rbx, %rbp, %r12, %r13, %r14, %r15.
```

OptiJava uses its own calling convention mechanism for Java methods. It allows flexibility in the allocation of registers to a method's parameters as well as caller-save and callee-save registers. The convention is applied over a method group, hence, the registers used for passing the parameters, caller-save registers and callee-save registers are common to all the methods within that method group.

For native methods, OptiJava uses the standard calling convention. In a method group that has any native methods, the register convention used is according to the convention stated by the ABI of the underlying architecture and all the methods in that method group will follow the standard convention. If the called method wants to interact with the native code, it should follow the native code's calling convention. The registers used for passing the parameters to the native method, the register used for passing the result of the method, the caller-save and the callee-save registers will be as specified in the ABI. To reflect the registers used for parameters and the register used for return, the basic block created for native method known as native basic block will have the operands of the entry values and exit values as fixed registers. The fixed registers assigned will be in accordance with the ABI specification.

For example, consider a C code that takes in 3 integer parameters and returns an integer result, then the native basic block created for this C code will have its operands of the entry and exit values as :

```
Operands of entry values : %rdi %rsi %rdx
Operands of exit value   : %rax
```

## 3.7 Conversion to Assembly

The final phase of OptiJava is the assembly phase that emits the assembly code (native machine code). An assembly instruction includes an operation on source registers and an assignment of the result to a destination register. Assembly instructions in general is of two types: 2-register instructions 3-register instructions

In 2-register instructions, the maximum number of registers that can be in the instruction is limited to two. Here, one of the source registers acts as the destination register as well. For example:

```
add r1, r2
```

Here, the value in register r1 is added with the value in register r2 and the result is stored in register r2.

In 3-register instructions, there can be three registers in the instruction, operator operates upon two source registers and the result is assigned to a destination register. For example;

```
add r1, r2, r3
```

Here, the value in register r1 is added with the value in register r2 and the result is stored in register r3.

The instruction format depends on the underlying architecture. Thus, OptiJava produces 3-register instructions if the target machine supports 3-register instructions and 2-register instructions otherwise.

Conversion to assembly is a two-step process; linearization of basic blocks and conversion of the IR instructions to assembly instructions. Linearization and conversion are done in parallel.

## 3.7.1 Linearization of blocks

When a program is run, the processor always checks the cache to fetch the next instruction. Caches are fast on-chip memories that are used to store frequently accessed instructions and data from main memory. If the needed information has to be fetched from main memory, many CPU cycles are lost where no computation can occur. A cache miss is a failed attempt to fetch the instruction from the cache and can generally cause the largest delay because the processor has to wait (stall) until the instruction is fetched from main memory. [33]

Linearizing the blocks attempts to avoid cache misses of frequently executed code. The assembly code layout is arranged in a manner where related pieces of code are placed close to each other, branches to highly likely exectued code is placed immediately after the block. Lineraization arranges the blocks so that a basic block is followed by the most likely block. That is, the fall through code or in other words the next sequential line of code where execution would continue is always placed next to each other.

In case of a basic block ending in a conditional branch instruction, the basic block has two successor blocks. "Many processors have asymmetric branch costs; the cost of a fall-through branch is less than the cost of a taken branch".[3] If it is possible to predict the direction that a branch instruction might take, compiler can choose which block to lie on the fall-through path and which block to place in the taken path.[32] The goal of OptiJava is to work along with the hardware's branch prediction capability.

Profile information helps in accurate branch prediction. The profile information will have the information of the frequently executed paths. Using such information, it can predict the probable outcome of conditionals.[9] Thus, OptiJava will place the highly likely executed branch path to be the fall through branch.

The steps for linearization is as follows:

1. First block of the method is placed in a queue.
2. The first block is removed from the queue and becomes the current block for linearization. The IR instructions of the current block is converted to assembly instructions.
3. Next, the first target block of the current block gets added to the front of the queue and any additional target blocks are added to the end of the queue.
4. Steps 2 and 3 are repeated until the queue is empty.

The above process is repeated for all the referenced methods.

Consider for example the control flow graph of method code as shown in Figure 3.11. The linearized order of the basic blocks of the method code is as shown in Figure 3.23.

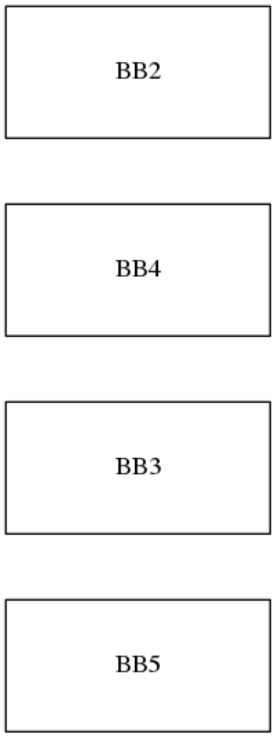


Figure 3.23: Linearized order of the basic blocks of method code

The IR instructions of the basic block are converted to assembly instructions in sequence. The assembly instruction is generated by emitting the assembly opcode of the IR instruction's opcode and computing the fixed register operands of the IR instruction's from-values and to-value.

The instructions are converted to assembly in sequential order. This is a three step process:

1. Evaluate the register operands that were assigned to the `find()` of the values.
2. Evaluate the fixed register mapping of the register operands. If the operand assigned was a generic register and if this generic register did not have any mapping to any fixed register after coagulation then any unmapped fixed register is mapped to this generic register.
3. Select an efficient translation of the IR instruction to assembly instruction.

For example, the IR instructions of the basic block BB2 of the method code is as shown in Figure 3.21 are:

0: `ILOAD 1,r7bb2`

1: `ILOAD 3,r9bb2`

2: `IADD r7bb2,r9bb2,r5bb2`

3: `COMPARE r5bb2,r0bb2,cc1bb2`

4: `IF_ICMPLE cc1bb2`

The conversion of these IR instructions to assembly would be as follows:

0: `ILOAD 1,r7bb2`

```
Find() of r7bb2                     : r7bb2
Operand assigned                    : G2
Fixed register mapping of G2        : null
New fixed register mapping for G2   : %rax
Assembly mnemonic                   : mov
Assembly instruction                : mov $1, %rbx
Assembly instruction selected       : xor %ebx,%ebx and inc %ebx
```

The actual conversion of the load instruction would be “mov $1, %rbx”. However, we select assembly instruction (“inc”), which would result in reduced cache line for the load instruction. The xor intruction clears the register ebx(four bytes of register rbx) which results in %ebx having the value 0. inc increments the value stored in %ebx.

1: `ILOAD 3,r9bb2`

```
Find() of r9bb2                    : r9bb2
Operand assigned                   : G3
Fixed register mapping of G3       : null
New fixed register mapping for G3 : %rcx
Assembly mnemonic                  : mov
Final assembly instruction         : mov $3, %rcx
```

2: `IADD r7bb2,r9bb2,r5bb2`

```
Find() of r20bb2                   : r7bb2
Operand assigned for r7bb2         : G2
Fixed register mapping of G2       : %rbx
Find() of r9bb2                    : r9bb2
Operand assigned for r9bb2         : G3
Fixed register mapping of G3       : %rcx
Find() of r5bb2                    : r5bb2
Operand assigned for r5bb2         : G3
Fixed register mapping of G3       : %rcx
Assembly mnemonic                  : %rcx
Assembly instruction               : add %rbx,%rcx,%rcx
Two register assembly instruction : add %rbx,%rcx
```

3: `COMPARE r5bb2,r0bb2,cc1bb2`

```
Find() of r5bb2                : r5bb2
Operand assigned for r5bb2     : G3
Fixed register mapping of G3 : %rcx
Find() of r0bb2                : r0bb2
Operand assigned for r0bb2     : G1
Fixed register mapping of G1 : %rdi
Find() of cc1bb2               : cc1bb2
Operand assigned for r0bb2     : G4
Fixed register mapping of G4 : %rdx
Assembly mnemonic              : cmp
Assembly instruction           : cmp %rdi,%rcx
```

4: `IF_ICMPLE cc1bb2`

```
Find() of cc1bb2                  : cc1bb2
Operand assigned for r0bb2        : G4
Fixed register mapping of G4      : %rdx
Assembly mnemonic                 : jle
Final assembly instruction        : jle bb4
```

All the branch instructions are converted to jump to second target of the basic block. This branch instruction would be "jle bb4" (basic block BB4 is the second target of the block BB2).

In case of instructions which require their values to access the stack or memory, such as spill instruction, unspill instruction and virtual call instruction, the offsets are computed and assembly instruction is generated accordingly. While converting a spill instruction, the offset of spill location on the stack with respect to the stack pointer is calculated. While converting a call instruction to a virtual method, the method offset is calculated from the virtual dispatch table. The process of conversion is repeated until all the IR instructions of the basic block is converted to assembly instruction.

The assembly process ends once all the instructions of all the generated basic blocks are converted to assembly instructions. The assembly output is printed out to a file which has the same name as the class file which was used as the input to the compiler. Labels for each method are added in the assembly output. A method lable consists of the name of the package , method name, parameter signature and the return signature. Method labels are of the format :
`_package-name_method-name__parameter-signature"r"return-signature`. This marks the end of compilation. The final assembly code of the RunningExample program is as shown in Figure 3.24.[2]

## 3.7.2 Compiler/Run-Time data structures

The data structures, data layout for objects and arrays, initialization and support routines are described in this section.

### Frequency table

OptiJava uses profiling to gather information about frequently executed paths of the code. OptiJava uses this frequency information to recompile the class files to produce efficient native code. Frequency table is a data structure used by OptiJava to cache

[2]the assembly code shown here does not include the code for class initializer and constructors

```
_test_RunningExample_main__aLjava_lang_String__rV:
# Parameters: (%rax)
bb1: movq $3,%rax
     movq _test_L000B_z(%rip), %rdi
     add %rax,%rdi
     mov %rdi, spill1(%rsp)
     call _test_RunningExample_code__IrI
bb6: mov %rdi, spill2(%rsp)
     call _java_lang_System_exit__IrV
bb7: ret
_test_L000B_code__IrI:
# Parameters: (%rdi)
bb2: xor %ebx,%ebx
     inc %ebx
     movq $3,%rcx
     add %rbx,%rcx
     cmp %rdi,%rcx
     jle bb3
bb4: movq $4,%rbx
     mullw %rdi,%rbx
bb5: add %rbx,%rdi
     ret
bb3:
cbb1:mov %rdi,%rbx
     jmp bb5
```

Figure 3.24: Assembly code of the the RunningExample program

the profile information. The execution frequency information of the edges are stored in this frequency table. That is, the count of the number of times the edge was executed is stored in this table. To increment the count of a particular edge, the count of the edge is first accessed by offsetting to the table and the value stored at that offset is incremented by 1. Each entry in the frequency table occupies 8 bytes. The edge-id gives the offset to the table. The frequency information of the edge whose id is 1 is accessed by offsetting 8(1*8) bytes from the base of the frequency table. Frequency information of the edge whose id is 5 is accessed by offseting 40(5*8) bytes from the base of the frequency table. The frequency information of call-edges of a block are grouped together. That is, consider a method which was called by two different methods. The first block of the called method will have two in-edges each representing the call-edge from its respective caller. In such a scenario, the frequency information of both the in-edges are placed together. This grouping of the in-edges is to assist in updating the call-edges properly. Each of the caller will load an offset which is specific to that caller to a reserved register. The called method will take the offset from the reserved register to additionally offset in the group of edges to update the information.

#### Objects, Class objects and Arrays

Every object has a pointer to its class as its first field. The remaining fields contain the object's instance variables.

The class pointer points to an instance of the java.lang.Class class. As this is an object it starts with a pointer to its class: java.lang.Class. This is followed by heap size of an instanceof the class, a pointer to the field descriptor for instances, a pointer to its super class, and a String containing the name of the class. Finally there is a virtual function table – a series of pointers to the code for each non-static method.

Data layout of an array consists of the type of the array as its first field. The second field is the length of the array. The rest of the object is the sequence of the elements of the array, starting at element 0.

### 3.7.3 Initialization and Setup

#### Initialization

- A C code called main.c is called from the operating system.
- main.c produces the parameter for the main of the designated class.
- It saves and clears all the physical registers of the underlying architecture.

- It then calls the method "`optiJava_main`".
- `optiJava_main` is a wrapper function to the main method of the designated class. The argument to this method is a reference to an instance of java.lang.String.
- `optiJava_main` calls the class initializers to initialize the classes.
- `optiJava_main` then calls the main of the designated class.

**Support routines**

A Java compiler must have an implementation of the standard java class libraries. The Java Class Library is a set of dynamically loadable libraries that Java applications can call at run time. The goal of OptiJava is to connect directly to existing class libraries. We implicitly use the class library that is implemented in the system. At this stage, OptiJava implements few classes of the library that overrides the existing classes. The logic of the overriden classes has been implemented in Java; native code is used only to call the necessary system functions. OptiJava implements the following classes:

- `java.io.PrintStream`
- `java.lang.Integer`
- `java.lang.Object`
- `java.lang.String`
- `java.lang.System`
- `java.lang.Throwable`

## 3.8 Benchmarks

In this section, the benchmark used in evaluating the performance of OptiJava is explained. The program as shown in Figure 3.25 was chosen as the benchmark. This benchmark was chosen because it enabled to compare the performance of compilers on core features of Java such as objects, method dispatch, and basic operations as well as to show the performance of compilers compiling programs having frequently executed parts of code.

This program has four loops,a simple loop which does an arithematic operation, a static loop which calls a static method, a virtual loop which calls a virtual method and a double loop which creates two objects and calls the virtual method twice. The number

```
public class Loops {
    static int timestamp=(int)System.nanoTime();
    static final int iterations = 5000000;
    private static void timing(String test) {
        System.err.print("Run␣for␣");
        System.err.print(test);
        System.err.print((int)System.nanoTime()-timestamp);
        System.err.println("ns");
        timestamp=(int)System.nanoTime();
    }
    private static int a(int p) {
        return p&1;
    }
    private static int c(Loops l,int p) {
        return l.b(p);
    }
    private int b(int p) {
        return p&1;
    }
    public static void main(String[] args) {
        int result=0, res = 0;
        for(int i=0;i<iterations;++i)
            result = result + 1;
        timing("Simple␣loop");
        result=0;
        for(int i=0;i<iterations;++i)
            result = result + a(i) + a(result);
        timing("Static␣call␣loop");
        result=0;
        Loops l = new Loops();
        for(int i=0;i<iterations;++i)
            result = result + l.b(i) + l.b(result);
        timing("Virtual␣call␣loop");
        result=0;
        Loops l2 = new Loops();
        for(int i=0;i<iterations;++i)
            result = result + c(l,i) + c(l2,result);
        timing("Double␣call␣loop");
    }
}
```

Figure 3.25: Java source code for benchmark

of iterations for each loop is 10000000. The benchmark program was run 10 times. This was done to allow adaptice compilers to reach a steady state.

For this benchmark, OptiJava compiled the class file and the generated native code was executed to evaluate its execution time. The frequency information generated after executing the native code was then used by OptiJava to recompile the class file. This newly generated native is then executed to evaluate the execution time. Hence, we have two execution times of OptiJava, one without frequency and one with frequency information. The overall run times were measured with the timestamp system utility.

### 3.8.1 Comparator environments

The compilers that have been used for comparison are:

1. Excelsior JET
2. GCJ
3. HotSpot JVM (64-Bit Server VM (build 25.151-b12, mixed mode)

The environment used is a Linux operating system with 64-bit architecture.

### 3.8.2 Test reporting

The results have been reported by comparing performance of OptiJava with respect to other compilers and comparing performance of OptiJava with profiling with respect to othjer compilers.

## 3.9 Limitations and Risks to Experimental Validity

- OptiJava supports only single threaded applications.
- Garbage collection has not been integrated.
- Does not support dynamic loading.

# Chapter 4

# Results

In this chapter, we discuss the performance of OptiJava. We also compare the performance of OptiJava with respect to other Java systems. The performance was evaluated using execution time as the criteria. The measurements were taken by executing the benchmark program by each of the comparator Java systems.

## 4.1 Performance of OptiJava and other Java Systems

The performances of OptiJava with and without frequency information, HotSpot JVM, Excelsior JET and GCJ compilers are examined in this section. We analyze the execution time taken by a compiler for each of the loops of the chosen benchmark program as shown in Figure 3.25. OptiJava with frequency information is shown as OptiJavaPGO. The execution time taken by OptiJava, OptiJavaPGO, HotSpot JVM, Excelsior JET and GCJ for each of the loops of the program mentioned in Figure 3.25 is as shown in 4.1. The time taken is shown in nanoseconds.

| **Java System** | **Simple Loop** | **Static Loop** | **Virtual Loop** | **Double Loop** |
|---|---|---|---|---|
| Excelsior JET | 10,859,406 | 80,017,237 | 70,517,584 | 133,340,723 |
| GCJ | 6,480,227 | 77,998,135 | 86,373,260 | 173,104,054 |
| HotSpot JVM | 3,530,851 | 13,545,983 | 13,932,022 | 15,208,217 |
| OptiJava | 5,686,204 | 47,926,494 | 58,845,157 | 83,507,569 |
| OptiJavaPGO | 7,802,540 | 55,135,895 | 59,407,229 | 84,943,360 |

Table 4.1: Execution time taken by the Java systems

## 4.2 Comparisons

The comparison results are examined in this section. To compare the results, the execution time taken by OptiJava is taken as the standard. The execution time of OptiJavaPGO, HotSpot JVM, Excelsior JET and GCJ are shown as percentage relative to OptiJava as shown in 4.2 . From the table we can see that OptiJava performed better

| **Java System** | **Simple Loop** | **Static Loop** | **Virtual Loop** | **Double Loop** |
|---|---|---|---|---|
| Excelsior JET | 190% | 167% | 120% | 160% |
| GCJ | 114% | 163% | 147% | 207% |
| HotSpot JVM | 62% | 28% | 24% | 18% |
| OptiJava | 100% | 100% | 100% | 100% |
| OptiJavaPGO | 137% | 115% | 101% | 102% |

Table 4.2: Execution time relative to OptiJava (smaller is better)

than the AOT compilers Excelsior JET and GCJ. However, its performance was not satisfactory compared to HotSpot JVM.

## 4.3 Analysis

These results indicate that both OptiJava and OptiJava with profiling produced an executable whose performance was superior to other commercial AOT compilers. However, OptiJava is slower than Oracle's reference Just-In-Time compiler.

A surprising factor was regarding the performance of OptiJava with profiling. The time taken to execute the benchmark program by OptiJava with frequency information was slower than the time taken to execute using OptiJava without the frequency information. We expected that OptiJava with profiling would perform better because it had information of the frequently executed parts of the code. However, to our disappointment it fared worse than without profiling. We believe that this could be because of excessive register shuffling happening at the lower frequent edges, and will be looking at resolving this in the very near future!

# Chapter 5

# Conclusions

We have presented the design and implementation of OptiJava; a native-code Ahead-Of-Time compiler that converts `.class` files to exeutable code. OptiJava is implemented completely in Java. We have focussed more on providing an efficient code which would reduce the execution time rather than on the compilation speed. The discussion in this dissertation has been focussed on x86-64 architecture with Linux operating system. However, the techniques are common to other platforms as well.

The Intermediate Representation (IR) used in OptiJava is a strongly typed, register based and Static Single Assignment (SSA) representation. Generally, compilers that use SSA based IR, use a time consuming technique to convert the IR to SSA based IR. However such a transforamtion is not necessary in OptiJava because SSA form is implict in the OptiJava's IR instructions.

Unlike traditional compilers which focuss their optimizations on method level, OptiJava concentrates on optimizing the busiest/hot parts of the code. This stems from the fact that the most of the execution time spent by an application is on loops or recursive calls. OptiJava gives importance to these parts of the program by allocating enough registers for its usage thereby aiming to reduce the execution time. This is done by using the concept of "code coagulation" as its compilation technique. Coagulation ensures that the first regions compiled will be a program's busy loops. Coagulation uses a run-time profile for the program being compiled for code generation. By treating busy parts of a program first and using the strategy of local optimiality, OptiJava maximises the benefit of careful instruction selection, register allocation and interprocedural optimization while avoiding unnecessary data movement in busy sections.

The main contributions in this dissertation are:

- Implementation of the coagualtion concept in an abstract virtual machine.

- In the past, the coagulation technique has only been implemented in a C compiler. Through this research, we have implemented the coagulation concept in an object oriented language for the first time.
- Generally, compilers convert intermediate instructions to be in a Static Single Assignment (SSA) form. This process of conversion is time consuming. We have introduced a technique where the intermediate instructions can be of SSA form implicitly without having to carry any additional conversion process.

In our limited tests, the results prove that OptiJava achieved superior performance to existing AOT Java compilers. Although OptiJava is slower than the HotSpot Just-In-Time JVM, we believe that there is considerable scope for improvement in OptiJava.

## 5.1 Future Work

We are working to bring the OptiJava to a production state and improve its performance. This includes:

- Preliminary assessment indicates that reshuffling of registers can be better handled.
- Register spills can be reduced by improving the register allocation algorithm.
- Native code can be optimized by using object-oriented optimizations for handling the array out-of-bound checks, null pointer exeception checks and method inling.
- Improve the exception handling mechanism by finding the matching catch block in the entire program rather than confining the search within the current method and also by integrating runtime environment handle.
- Integrate garbage collection and multi-threading.
- Evaluate the performance of OptiJava using standard benchmarks.

More general future work includes:

- Another challenging research would be to support dynamic loading. Our thought is to include an interpreter in the code, which would benefit from the coagulation.
- Currently OptiJava supports only the x86-64 architecture with Linux and Mac-OS-X operating systems and we want to port it to different architectures and different operating systems. The code for this is already in place and all that is left to be done is to integrate it to the register allocation and assembly phase of OptiJava.

# Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Don Mills, Ontario: Addison-Wesley, 2007. ISBN: 0-201-10088-6.

[2] B. Alpern et al. "The Jikes Research Virtual Machine Project: Building an Open-source Research Community". In: *IBM Syst. J.* 44.2 (Jan. 2005), pp. 399–417. ISSN: 0018-8670. DOI: 10.1147/sj.442.0399.

[3] Anklam. *Engineering a Compiler.* Digital Press, 1982. ISBN: 0-932376-19-3.

[4] John Aycock. "A Brief History of Just-in-time". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

[5] Per Bothner. "Compiling Java with GCJ". In: *Linux J.* 2003.105 (Jan. 2003), pp. 4–. ISSN: 1075-3583.

[6] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. "Integrating Register Allocation and Instruction Scheduling for RISCs". In: *SIGPLAN Not.* 26.4 (Apr. 1991), pp. 122–131. ISSN: 0362-1340. DOI: 10.1145/106973.106986.

[7] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. "The Jalapeño Dynamic Optimizing Compiler for Java". In: *Proceedings of the ACM 1999 Conference on Java Grande.* JAVA '99. San Francisco, California, USA: ACM, 1999, pp. 129–141. ISBN: 1-58113-161-5. DOI: 10.1145/304065.304113.

[8] G. J. Chaitin. "Register Allocation & Spilling via Graph Coloring". In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction.* SIGPLAN '82. Boston, Massachusetts, USA: ACM, 1982, pp. 98–105. ISBN: 0-89791-074-5. DOI: 10.1145/800230.806984.

[9] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. “Using Profile Information to Assist Classic Code Optimizations”. In: *Softw. Pract. Exper.* 21.12 (Dec. 1991), pp. 1301–1321. ISSN: 0038-0644. DOI: `10.1002/spe.4380211204`.

[10] Keith D. Cooper and L. Taylor Simpson. “Live Range Splitting in a Graph Coloring Register Allocator”. In: *Proceedings of the 7th International Conference on Compiler Construction.* CC ’98. London, UK, UK: Springer-Verlag, 1998, pp. 174–187. ISBN: 3-540-64304-4.

[11] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. “Compiling Java Just in Time”. In: *IEEE Micro* 17.3 (May 1997), pp. 36–43. ISSN: 0272-1732. DOI: `10.1109/40.591653`.

[12] Yuxin Ding, Jia Mei, and Hu Cheng. “Design and implementation of Java just-in-time compiler”. In: *Journal of Computer Science and Technology* 15.6 (Nov. 2000), pp. 584–590. ISSN: 1860-4749. DOI: `10.1007/BF02948840`.

[13] M. Anton Ertl and David Gregg. “The Structure and Performance of Efficient Interpreters”. In: *Journal of Instruction-Level Parallelism* 5 (2003), p. 2003.

[14] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. “Marmot: An Optimizing Compiler for Java”. In: *Softw. Pract. Exper.* 30.3 (Mar. 2000), pp. 199–232. ISSN: 0038-0644. DOI: `10.1002/(SICI)1097-024X(200003)30:3<199::AID-SPE296>3.0.CO;2-2`.

[15] *GCC Wiki - GCJ.* Accessed: 2018-01-08. URL: `https://gcc.gnu.org/wiki/GCJ`.

[16] John L. Hennessy and Thomas Gross. “Postpass Code Optimization of Pipeline Constraints”. In: *ACM Trans. Program. Lang. Syst.* 5.3 (July 1983), pp. 422–448. ISSN: 0164-0925. DOI: `10.1145/2166.357217`.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 978-0123838728.

[18] C. H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. “Java bytecode to native code translation: the Caffeine prototype and preliminary results”. In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29.* Dec. 1996, pp. 90–97. DOI: `10.1109/MICRO.1996.566453`.

[19] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. “Annotating the Java bytecodes in support of optimization”. In: *Concurrency: Practice and Experience* 9.11 (), pp. 1003–1016. DOI: `10.1002/(SICI)1096-9128(199711)9:11<1003::AID-CPE346>3.0.CO;2-G`.

[20] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. "Design, Implementation, and Evaluation of Optimizations in a Just-in-time Compiler". In: *Proceedings of the ACM 1999 Conference on Java Grande.* JAVA '99. San Francisco, California, USA: ACM, 1999, pp. 119–128. ISBN: 1-58113-161-5. DOI: `10.1145/304065.304111`.

[21] Michael Karr. "Code Generation by Coagulation". In: *Conference Record of the 1984 ACM SIGPLAN Symposium on Compiler Construction.* Vol. 19. 6. Association for Computing Machinery, June 1984, pp. 1–12.

[22] Michael Karr, Walter G Morris, and Steve Rozen. *Global Optimization for a Coagulating Code Generator: Final Technical Report*.* Tech. rep. 1991. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.7522`.

[23] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition.* 1st. Addison-Wesley Professional, 2013. ISBN: 978-0133260441.

[24] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. "System V Application Binary Interface". In: *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[25] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin. "Overview of Excelsior JET, a High Performance Alternative to Java Virtual Machines". In: *Proceedings of the 3rd International Workshop on Software and Performance.* WOSP '02. Rome, Italy: ACM, 2002, pp. 104–113. ISBN: 1-58113-563-7. DOI: `10.1145/584369.584387`.

[26] W. G. Morris. "CCG: A Prototype Coagulating Code Generator". In: *Conference Record of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation.* Vol. 28. 7. Association for Computing Machinery. Toronto, Canada, June 1991, pp. 45–58.

[27] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code". In: *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3.* COOTS'97. Portland, Oregon: USENIX Association, 1997, pp. 1–1.

[28] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java hotspotTM Server Compiler". In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Monterey, California: USENIX Association, 2001, pp. 1–1.

[29] Massimiliano Poletto and Vivek Sarkar. "Linear Scan Register Allocation". In: *ACM Trans. Program. Lang. Syst.* 21.5 (Sept. 1999), pp. 895–913. ISSN: 0164-0925. DOI: `10.1145/330249.330250`.

[30] Todd A. Proebsting, Gregg Townsend, John H. Hartman, Patrick Bridges, Scott A. Watterson, and Tim Newsham. "Toba: Java for Applications A Way Ahead of Time (WAT) Compiler". In: Advanced Computing Systems Association, June 1997.

[31] Ramesh Radhakrishnan, R. Radhakrishnany, Lizy K. John, Juan Rubio, L. K. Johny, and N. Vijaykrishnan. *Execution Characteristics of Just-In-Time Compilers.* 1999. DOI: `10.1.1.44.2640`.

[32] James E. Smith. "A Study of Branch Prediction Strategies". In: *Proceedings of the 8th Annual Symposium on Computer Architecture.* ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 135–148.

[33] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition.* 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2007. ISBN: 978-1420043822.

[34] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. "Overview of the IBM Java Just-in-time Compiler". In: *IBM Syst. J.* 39.1 (Jan. 2000), pp. 175–193. ISSN: 0018-8670. DOI: `10.1147/sj.391.0175`.

[35] *The Java HotSpot Performance Engine Architecture.* Accessed: 2018-01-08. URL: `http://www.oracle.com/technetwork/java/whitepaper-135217.html#3`.

[36] Bill Venners. *Inside the Java Virtual Machine.* New York, NY, USA: McGraw-Hill, Inc., 1996. ISBN: 0079132480.

[37] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler. "TurboJ, a Java bytecode-to-native compiler". In: *Languages, Compilers, and Tools for Embedded Systems: ACM SIGPLAN Workshop LCTES'98 Montreal, Canada, June 19–20, 1998 Proceedings.* Ed. by Frank Mueller and Azer Bestavros.

Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 119–130. ISBN: 978-3-540-49673-1. DOI: `10.1007/BFb0057785`.