JVM COMPILER BACKEND FOR BALLERINA INTERMEDIATE REPRESENTATION

Thangarajah Kishanthan

179329D

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2019

JVM COMPILER BACKEND FOR BALLERINA INTERMEDIATE REPRESENTATION

Thangarajah Kishanthan

179329D

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2019

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

(T. Kishanthan)

The above candidate has carried out research for the Masters thesis under my supervision.

Signature: Date:

(Dr. Indika Perera)

ABSTRACT

Ballerina is an open source, strongly typed language for writing microservices and network applications with main focus on solving enterprise integration requirements. The ballerina compiler converts the ballerina source to set of ballerina byte code which is then executed by the ballerina virtual machine (BVM). The BVM does not perform well for most of the CPU bound operations due to its current design. This project focus on compiling ballerina source to JVM byte code and will be executed by the JVM directly, which will solve the performance bottleneck at BVM. This project also proposes a new compiler architecture, in which, the ballerina source code is transformed to an intermediate representation which is a low level representation of the ballerina program and it is used for generating the target JVM byte code. The performance of JVM based compiler backend implementation against the current BVM was compared for certain algorithms and programs. From the evaluation of the test results, it is found that the JVM target outperforms the ballerina runtime by factor of 100 in certain scenarios. With this promising results, the proposed new compiler architecture based on ballerina intermediate representation and the JVM compiler backend can potentially be used as the replacement for current ballerina compiler and runtime.

ACKNOWLEDGEMENT

I would like to express my profound gratitude to my supervisor Dr. Indika Perera, for his invaluable support throughout by advising and guiding me through the correct directions. His expertise and continuous guidance are one of the key reasons for the success of this research.

I should also thank Dr. Sanjiva Weerawarana (Founder and Chairman of WSO2) and Mr. Sameera Jayasoma (Senior Director at WSO2), who encouraged me to complete this research project. I am mostly thankful for my wife, my parents for their immense support and patience at all the time. At last but not least, I'm thankful to all my colleagues at WSO2, who helped me a lot in various ways throughout this research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	**Description**
BVM	Ballerina Virtual Machine
AST	Abstract Syntax Tree
IR	Intermediate Representation
BIR	Ballerina Intermediate Representation
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
CFG	Control Flow Graph

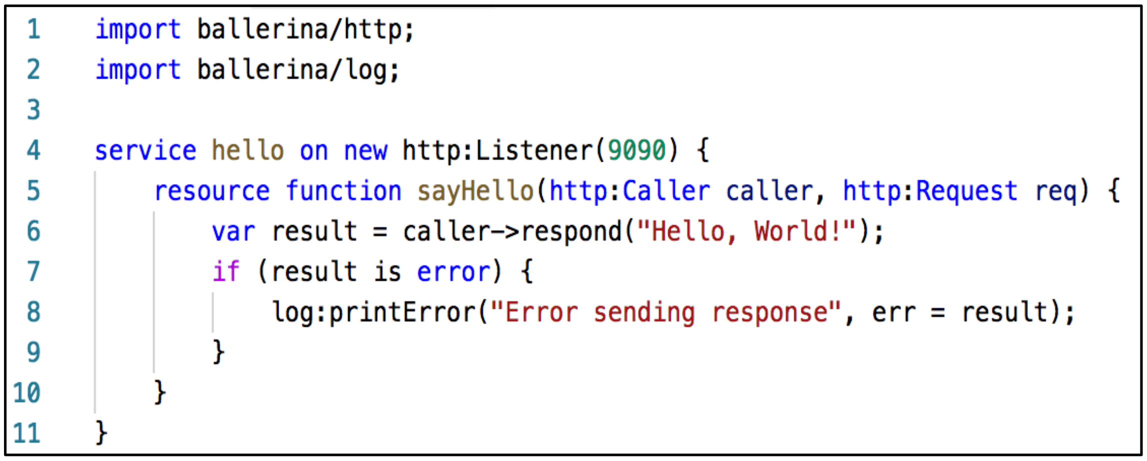
1. INTRODUCTION

The Introduction section is organized as follows. First, some basic concepts related to ballerina language, ballerina compiler and how the compiler is currently implemented are presented and then the problems in the existing compiler architecture are identified. At the end, it explains the motivation for this research project and concludes the section with objectives.

1.1 Ballerina

Ballerina [19] is a compiled and strongly typed programming language. It incorporates both textual and graphical syntaxes to write networked applications and microservices that exposes APIs. The main motivation for developing a new language is that it aims to fill the gap currently exists between integration systems and programming languages since it may require to write lot of boilerplate code with existing general purpose programming language to integrate systems and endpoints. On the other hand, ballerina provides first class support for well defined integration related constructs such as service, endpoints, and message types (json, xml, etc) which can used to write code with few lines.

Additionally, the graphical syntax of ballerina uses sequence diagrams to implement and visualize the microservices and APIs. Figure 1.1 shows the textual view and Figure 1.2 shows the graphical view based on sequence diagrams that is generated for the same ballerina program.



```
import ballerina/http;
import ballerina/log;

service hello on new http:Listener(9090) {
    resource function sayHello(http:Caller caller, http:Request req) {
        var result = caller->respond("Hello, World!");
        if (result is error) {
            log:printError("Error sending response", err = result);
        }
    }
}
```

Figure 1.1 Textual view of a ballerina program

The same ballerina program given in Figure 1.1 can be visualized using the ballerina IDE plugin in Figure 1.2.

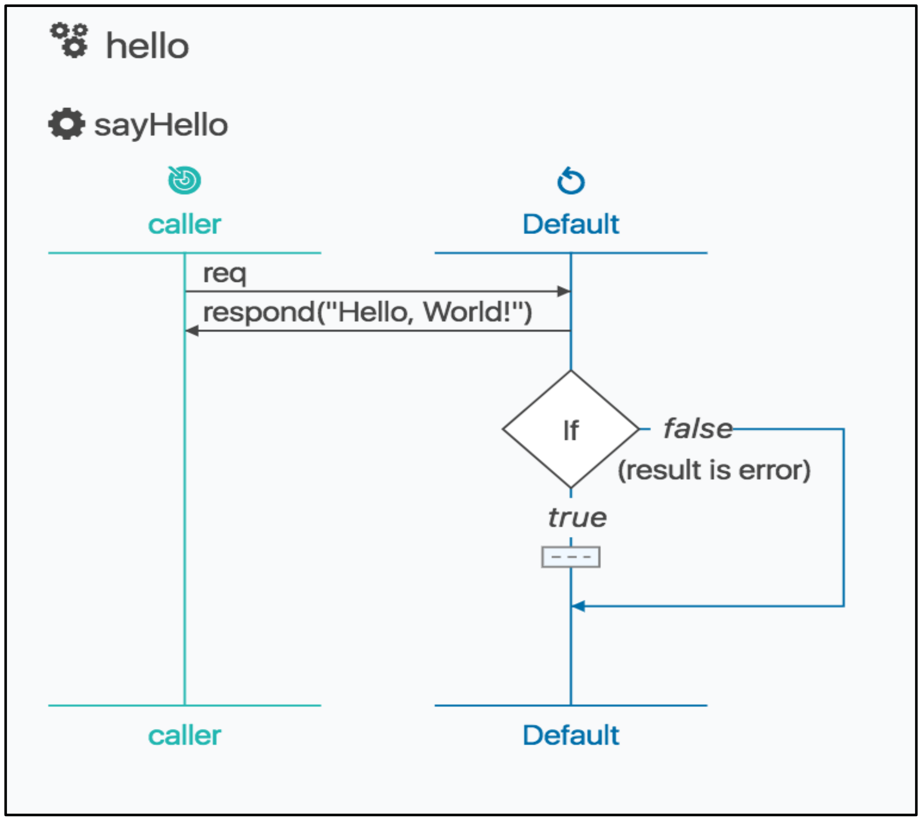


Figure 1.2 Graphical view of a ballerina program

1.2 Ballerina Compiler

Ballerina compiler is responsible for compiling the ballerina source to executable code. The compiler first transform the source to a tree representation known as the Abstract Syntax Tree (AST) [1]. This is a common step followed by any compiler to parse and transform source to a tree representation. This conversion to AST eases the operations and validation on it such as syntax validation and semantics validation. Once its validated, the AST is further analyzed to improve and remove any unwanted syntactic sugar coating, known as the desugar and code analysis phase. And finally, the AST will be converted to the ballerina bytecode, which is the executable form of the program.

A Ballerina program consists of one or more packages. A package consists of one or more Ballerina source files (.bal) and the package is the unit of compilation. The Ballerina compiler translates the source form of a package into its binary form. The binary form consists of set by bytecode instructions which will be interpreted and executed by the Ballerina runtime known as the Ballerina Virtual Machine (BVM). The current implementation of Ballerina compiler and runtime is written purely in Java. The current implementation which is based on Java serves as the reference implementation of the Ballerina specification [20].

Since Ballerina aims to solve the integration and microservice related problem, there can be various scenarios and use cases that will be written using Ballerina as a language. When considering application integration and writing microservices, there can be lot CPU and I/O bound operations that will be used by the programs written in Ballerina. For example, when we write a message transformation integration pattern [2], there will be lot of CPU bound operations that will be needed to do transforming the message content to various formats. Therefore performance of Ballerina runtime will play a key role for the success of the language and its adaptation among integration developers and companies.

1.3 Problem

Currently, compiling ballerina source files produces a binary file known as the ballerina bytecode. This compiled binary file is platform independent and it is interpreted and executed by the BVM runtime. The BVM is currently written in Java language. The performance of current BVM implementation is more than 100 times slower than of Java for most of the cases which have CPU bound operations. Since BVM is written as an interpreter, it has a thick layer which runs on top of the JVM. This introduces the significant performance drop compared to the Java. To solve this issue, the current BVM can be made as a very thin layer therefore it does not introduce any performance bottleneck. However there is a limit on how far we can proceed on this as BVM has to do the interpreting and execute. One other solution is to move away from BVM and compile ballerina source code to other compiled forms that can be executed by already available runtimes or compiling directly to machine native code.

Therefore the next stage in the Ballerina compiler and runtime evolution is to generate code that are directly understood by the underlying system and executed. This requires the ballerina source to be converted into a representation that is closely related to a machine code representation. However instead of directly generating machine code, the source can be first converted into an intermediate representation that can be further used for generating the target machine code.

In ballerina, the intermediate representation that will be generated for the source file is called Ballerina Intermediate Representation (BIR). Once a BIR is generated, it can be used for generating the target executable machine code. Since BIR will act as an intermediate representation which is a low level representation of the Abstract Syntax Tree (AST), it is independent from platform or operating system. Therefore any form of target code can possibly be generated from the BIR. For example, the BIR can be compiled to WebAssembly based bytecode [3] which can be executed by any web browsers. Likewise, the BIR can be compiled down to native code which can be directly executed.

1.4 Objectives

When considering the performance aspect, compiling the BIR to native code seems to be correct choice. However generating native code will incur lot of work such as implementing a memory management module (garbage collector), implementing a http library and other modules around it which is an important component needed for writing Ballerina based microservices and APIs. Considering these aspects, following are the identified objectives of this project.

- Generate compiled code for ballerina source which is directly executed by the underlying system which will remove the performance bottleneck of BVM.
- Convert Ballerina abstract syntax tree (AST) to some form of intermediate representation (IR), before the target executable code is generated. This will be called ballerina intermediate representation (BIR).
    - A portable, platform independent intermediate representation will make the code generation process easy with low level IR.
    - The BIR can be a common form for generating different target executable code (JVM, Native, WebAssembly, etc).
- Implement JVM compiler backend for BIR which will replace BVM and execute the ballerina program directly on JVM.
    - Existing java based frameworks and libraries can be directly used with JVM runtime.
    - Memory management is handled with Garbage Collector of JVM.
- Analyze the performance of JVM compiler backend against the existing BVM based runtime.
    - Compare performance of CPU bound operations and algorithms on both BVM based runtime and JVM backend.
    - Also analyze the existing performance issues and bottleneck with the BVM based runtime and propose possible improvements.

2. LITERATURE REVIEW

The Literature Review section is organized as follows. First, it analyze the existing compiler architecture of ballerina and the proposed new architecture. Then it analyzes the Ballerina Intermediate Representation and its requirements. Then it focus on how JVM is currently modeled, the structure of Java class binary format and how the JVM execute a Java class binary file which consists of bytecode instructions which are needed when modeling the BIR instructions to JVM instructions.

2.1 Ballerina Compiler

A program compiler is the tool that read, understand and transform a program source to another format called the bytecode. The bytecode is an intermediate representation of the source code and it will be most of the time, platform independent where the generated bytecode can be run on any other platforms (operating system) without the need to recompile the program again.

The ballerina compiler also is designed for the purpose of generating platform independent bytecode. Initial designs of the compiler was very simple where it followed a simple two phase of compilations. The compiler first reads and builds the abstract syntax tree (AST) representation and them analyze the AST for semantic correctness.. To build an AST representation, a compiler would need a parser that parses the given source code. Using the ANTLR grammar library [5], ballerina team wrote the grammar for the parser, which was then used to generate the source code parser. This parser was basically used with generating the AST tree. Then the compiler make sure that the language follows the correct semantics by analysis the generated AST.

Later with multiple features and language constructs being added, the above design of the compiler was not good enough. The current implementation of ballerina compiler is written in Java. As explained above, the current implementation of the compiler has a frontend (which does all the syntax and semantic validation and then generate the ballerina compiled byte code) and a backend which is the ballerina runtime implementation (BVM - Ballerina Virtual Machine) which is also written in Java. The choice of Java was to do a reference implementation of the Ballerina specification and the compiler and prove that the specification will work.

As the vision of the ballerina language to have fully fledged support for writing programs, the compiler design has to be further improved. The next step of compiler architecture evolution is to improve startup time with consuming less memory usage compared to the current Java based reference implementation. As with any modern language compilers, the ballerina compiler will also contain pipeline architecture with three separate compilation phases. The latest design of the ballerina compiler contains multiple phases where each of the phase is designed for a dedicated task in compilation the source as illustrated in Figure 2.1.

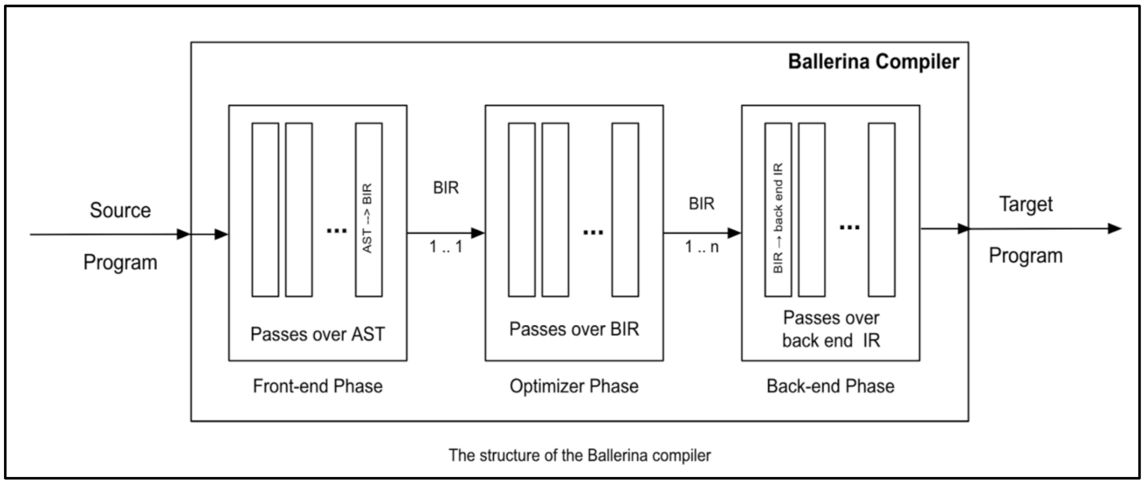


Figure 2.1 Ballerina Compiler Architecture [21]

Each of the phase lowers the source representation of the ballerina program to a more abstract representation with basic constructs therefore it becomes easy for the next phase inline to lower it further. Each phase makes sure that no information is lost from source when generating the target executable code. The separation of phases in compiler architecture is inspired from the LLVM compiler architecture [6] [9]. Let's look at these compiler phases in detail.

2.1.1 Frontend Phase

The front-end phase verify the syntax and semantics of the source and validate it. In this phase, the source will be first converted into a tree representation known as the Abstract Syntax Tree (AST) in order to ease the operations and validation on it. The validations in the front-end phase mainly focus on verifying first the syntax and then the semantics of the program. Various components such as type checker, semantic analyzer, symbol creator, etc at front end phase work together with validating the program for correctness as below.

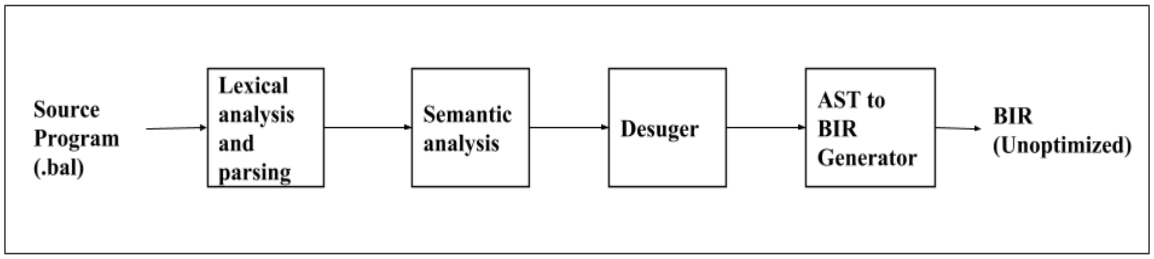


Figure 2.2 Ballerina Compiler Frontend

The high level passes using the above mentioned components, that compiler frontend undergo when compiling a ballerina source file, are as follows.

1. Lexical analysis and parsing
2. Semantic analysis
3. Code analysis
4. BIR generation

Each of the phase does one or more tasks in order to compile the source code correctly or fail when there is an error. Each phases and their tasks are examples as below.

Lexical analysis and parsing phase basically checks for the correct syntax with the source code and fails the compilation in the case of systemic errors found. It also builds the the AST tree that is used in the next step for analysis.

In order to analyse and parse a source code, a compiler needs to have a grammar defined [22]. A grammar is the base for form the language constructs according to the syntax and rules defined. Ballerina language grammar was written using the commonly and widely used ANTLR tool and the tool was used to generate the parser and the lexical analyser according the grammar defined. The parser will take the source code as input and basically parse (fire events) with the use of ANTLR event listeners which would then generate the AST eventually.

Semantic analysis is the semantic error analysis phase where the complete source code is analyzed for correct semantics. It mainly checks the symbols (variables, functions, structures. etc) and whether their defined in the correct scopes (block scope, functions scope, package scope, etc) and types (int, string. etc) and their correctness according the language definitions.

Code analysis phase basically checks correctness of the code such as unused imports, unreachable code segments, not returning functions. Etc. Additionally this also performs removal of any syntactic sugar that is supported at language level and remove them from the generated AST to generate a common structure that is used for code generation phase next.

The AST undergoes multiple passes with these components and gets enriched with information such as types, expressions, symbol definitions, etc along the way. After all the passes, the AST will be a validated tree representation of the source. The AST is more closer to the source, which will be harder for any machine to understand as it will contain too much information such as syntactic sugar [7]. And it will be harder to generate any low level machine code from the AST due to its nature. Therefore the AST now has to be converted into a more low level intermediate representation, which should be closer to the machine readable format. This is where the BIR generation comes into the picture. The BIR is considered closer to a machine readable format (such as a LLVM IR [8]). The conversion of ballerina AST to BIR will be the last pass in the front-end phase. The BIR contains only the basic set of language constructs such as while loops, variable declarations, if-else statements, function calls, etc. This makes it easy for generating any target executable as most of these basic constructs will be very similar in any of the target executable.

2.1.2 Optimizer Phase

The BIR version of the ballerina source will be fed into the optimizer phase of the compiler. In this phase, the optimizer will make multiple passes over the BIR and possibly optimizing it further by removing or replacing some instructions with more efficient instructions and avoiding time consuming paths, etc. This phase is most likely an analyzing phase as the BIR has to be refined and optimized further. The BIR is modeled as a control-flow-graph (CFG). Therefore using control flow algorithms and data flow analysis, the BIR can be further processed and optimized.

2.1.3 Backend Phase

This is a final phase of the compiler which converts the optimized BIR that comes from the optimizer phase to a low level target executable code. This phase will process and traverse through the BIR instructions and generate the target backend instruction set. The target executable generation can vary on the requirement.

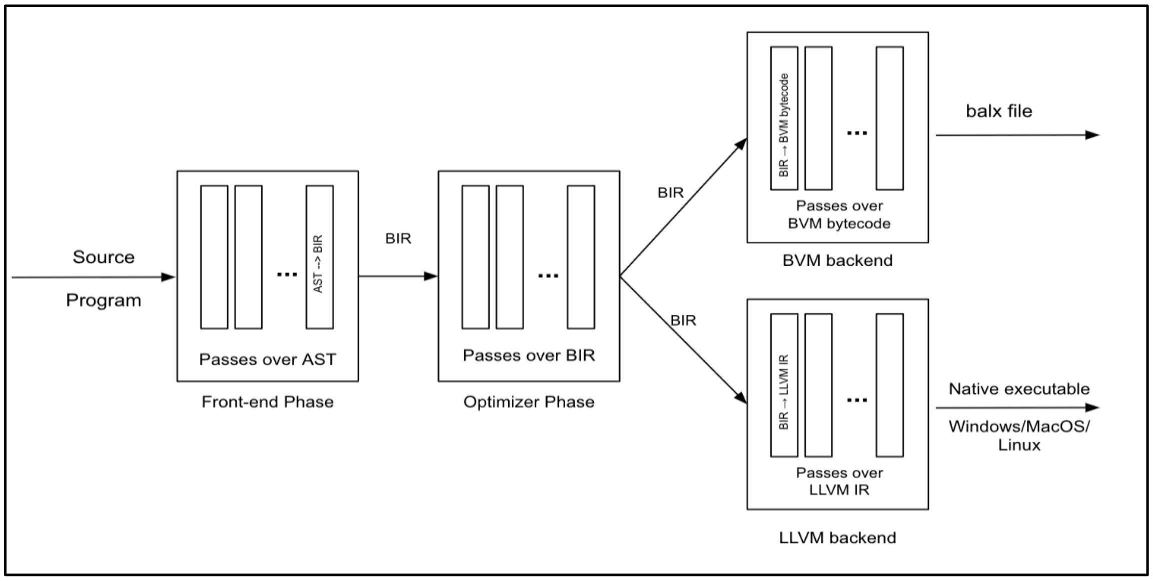


Figure 2.3 Ballerina Compiler Backend [21]

If the requirement is to generate the native executable on a target platform, then the backend phase has to use correct set of tools and logic to generate the native code. Likewise, there can be more than one target backend for ballerina compiler as given in the above. As described from the above diagram, we can create different targets for the BIR. The reference implementation which is based on the BVM will generate the ".balx" (ballerina byte code). The LLVM based target will generate native executable based on the OS. The JVM based compiler backend will generate the java bytecode (.class files) that can be executed by JVM.

2.2 Intermediate Representation

When a programming language evolves, it becomes more complex with syntax, and semantic rules. Therefore, all most all the modern compilers convert the source program into an intermediate representation (IR) [11] which contains the basic programming constructs, before compiling does to an executable format. This makes the target code generation easy for the compilers as the intermediate representation are generally low level, which are closer to machine understandable format than the AST of the program source, which is more closer to the source. An IR of a program is generally platform and hardware independent. Therefore it becomes easy for generating the target code by reading and processing this IR. For example, JVM bytecode [10] can be considered as an IR for the Java program and it is more low level and close to machine readable format. The JVM bytecode is also machine independent and it can be used with any platform and hardware. This IR is understood by the JVM and executed at the target machine.

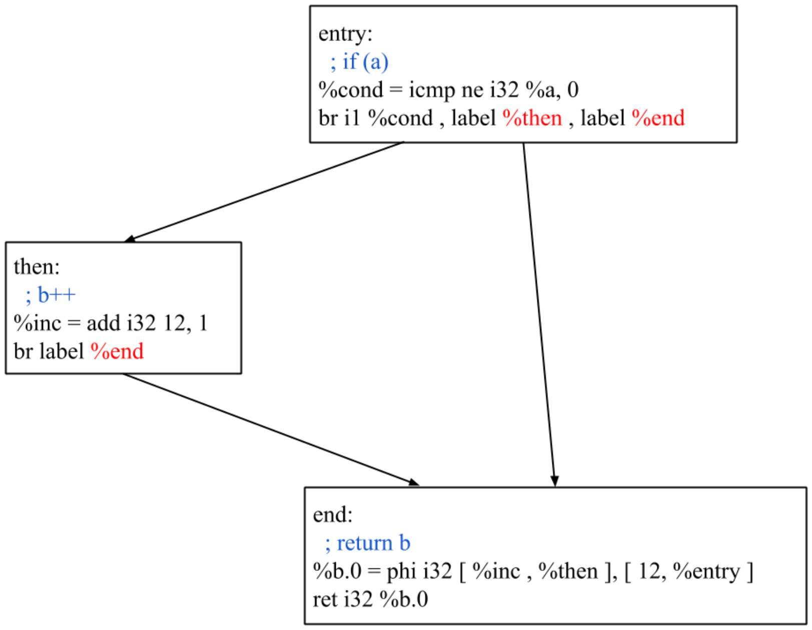
An IR is also independent of the source language. Therefore AST's from different types of languages can use the same IR as the intermediate state in the compiling phase. For example, the LLVM IR [9] is a uniform IR and any language can be converted to this IR which can be further compiled down to the native code. Similarly, the JVM bytecode also independent of the source therefore any programming languages can use JVM bytecode as their IR, which can be executed by the JVM runtime. The advantages of having a an IR which is independent of source programs (front-end) and target compiled executable (back-end) is that various different languages can be converted to the same IR and from the IR, various target executable can be generated. For example, for the same front-end, we can generate multiple backend based on the requirements.

An intermediate representation typically consists of basic language constructs only, such as, variable declarations, assignments, conditional branching, etc. And they are usually generated as a structured model. This is known as the control flow graph, which describes how the program control flow along various paths based on various conditions. Performing analysis and optimization on graph based model can be done using various techniques and algorithms such as loops detection, loop reduction, invariant code detection, code weight reduction, etc. Also the variables and values in an IR is represented either with stacks or registers. For example, the JVM bytecode IR and WebAssembly IR [3] uses stack based representation. LLVM IR is an example for register based IR. The main difference between a stack based IR and register based IR is that, when we want to execute an instruction, we will have to load the values (operands) on to the stack first, perform the operations and store the result back to another memory location. However with register based approach, the operations can be directly carried out on the registers which is usually the CPU registers, however we will need to know the register index locations. Both these implementation has their own advantages as explained in this paper [12].

As explained earlier, the IR is modeled as control flow graph (CFG), where the nodes are called basic blocks and the edges of the graph are how flow is traversed from one basic block to other basic blocks. A basic block is usually constructed by the compiler, which contains set of instructions that a grouped together as a single entity and it makes sure that all the instructions in the basic blocks executed completely. The execution of a basic block should complete full and once completing, it will traverse to the next basic block or terminate the execution if it was a method return instructions. The basic block execution could be interrupted only in the event of a system failure or hardware faults. Table 2.1 describes an example CFG for LLVM IR with some basic blocks. It also describes the conditional branching among basic blocks.

Table 2.1 Example LLVM CFG

Example Source	**Example IR as a CFG**

int b = 12;
if (a) {
  b++;
}
return b;
	
	 entry:
	; if (a)
	%cond = icmp ne i32 %a, 0
	br i1 %cond , label %then , label %end
	then:
	; b++
	%inc = add i32 12, 1
	br label %end
	end:
	; return b
	%b.0 = phi i32 [%inc , %then], [12, %entry]
	ret i32 %b.0

In this example, there are three basic blocks created at the IR level. Each of them has one or more instructions grouped together. Some of the basic blocks have branching and traverse instructions as the last instruction, which points to other basic blocks that continues the flow.

2.3 Ballerina Intermediate Representation

The ballerina intermediate representation is designed as a control flow graph using register based memory model for variable storage. The model of BIR is inspired from LLVM IR and SIL [22] based models. The functions in BIR represents the functions from the ballerina source and it's made up of one or more execution blocks known as the basic blocks. Each basic block consists one or more instruction which are the smallest unit of execution. Table 2.2 describes an example of a BIR that gets generated for ballerina source.

Table 2.2 Example BIR

Ballerina Source	**Example BIR**
function foo(int a) returns int {	function foo(int) -> int {
int b = 12;	int %0; // return
if (a > 0) {	int %1; // arg
b++;	int %2; // local
}	int %3; // temp
return b;	int %4; // temp
}	int %5; // temp
	boolean %6; // temp
	int %7; // temp
	int %8; // temp
	int %9; // temp
	int %10; // temp
	bb0 {
	%3 = const_load 12;
	%2 = %3;
	%4 = %1;
	%5 = const_load 0;
	%6 = greater_than %4 %5;
	branch %6 [true:bb1, false:bb2];
	}
	bb1 {
	%7 = %2;
	%8 = const_load 1;
	%9 = add %7 %8;
	%2 = %9;
	goto bb2;
	}
	bb2 {
	%10 = %2;
	%0 = %10;
	goto bb3;
	}
	bb3 {
	return;
	}
	}

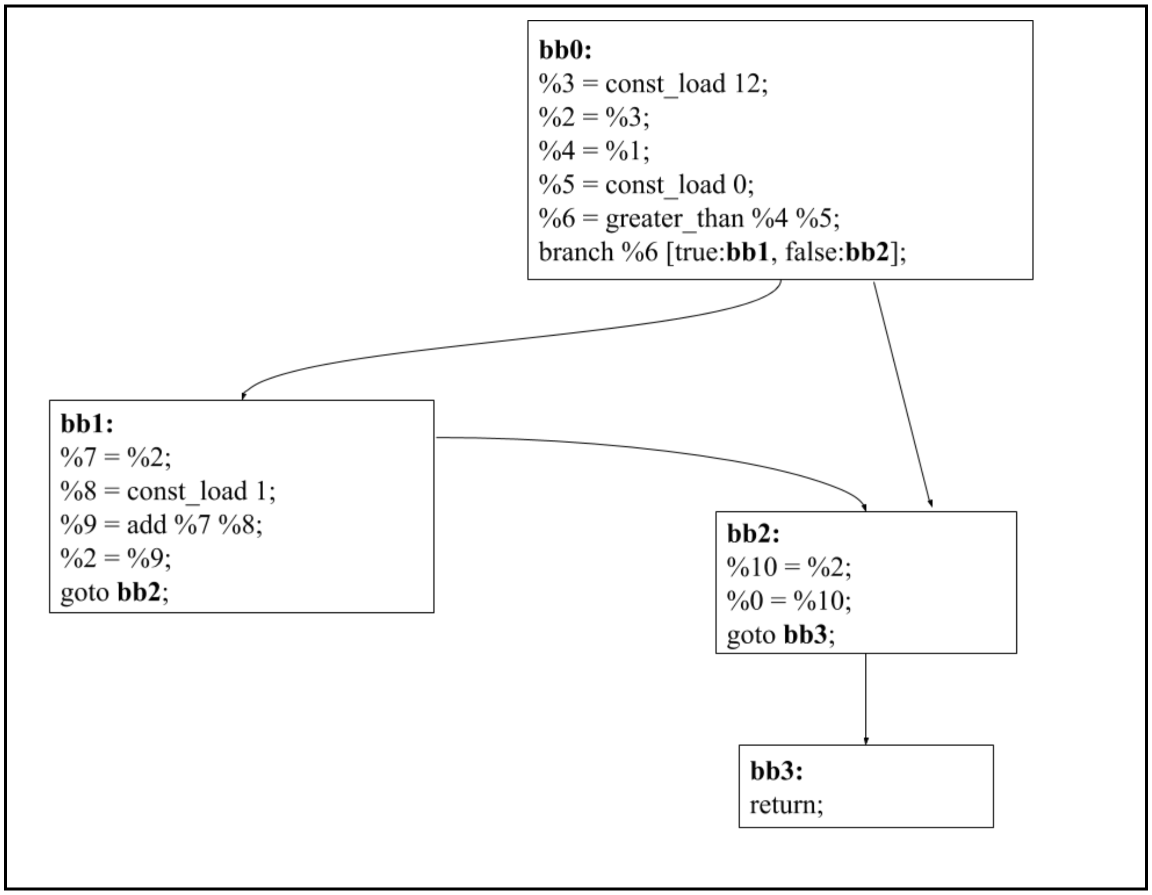


Figure 2.4 Example BIR in CFG

We can see that BIR is very similar to LLVM IR that was explained previously. Since BIR is a CFG, data flow and control flow analysis can be done on it to further optimize the graph. Each basic block contains set of instructions that are grouped such that they are executed together before the execution branches to another block. Termination of a basic block is usually a branching to another basic block or returning from the current function.

At high level, BIR model will have the constructs described in Table 2.3.

Table 2.3 BIR Constructs

BIR Construct	**Description**
Package	A package has a collection of functions with an organization

	name and a package identifier.
Function	A function has a set of basic blocks and s set of local variables with the visibility (public, private) flag along with the function identifier.
BasicBlock	A basic block has a set of instruction and a terminator along with a basic block identifier.
Name	Name is basically the identifier used by all the constructs in the BIR model
InstructionKind	The various types of instructions that are supported by the BIR. For example, Move, Call, Branch, Add, Equal, Greater_Than, etc.
VariableKind	Various types of variables supported by the BIR such as local, function argument, function return, and temporary (function local) variables.
VariableDeclaration	This represents the definition of a variable within a function. It contains the kind of the variable, and a type along with an identifier.
VariableReference	This represents a reference to other variables within the function. It contains the kind of the variable and type along the pointer to the variable it refers.
Types	The different types that are supported in BIR model currently. The types are, int, string, boolean, nil, array, etc.
Operands	The variable references that is used with each instruction as operands for execution.
Instructions	The smallest unit of execution in BIR. Each instruction has a

	kind and one or more operands associated with it. Some instruction also will have a reference to the basic block that it may need for traversing.

2.3.1 BIR Instructions

The instruction set of BIR describes how the program should get executed. Using these instruction set, the target executable code can be generated. The instruction set are basically set of low level instruction that are of machine readable format. Ttable 2.4 describes each instruction and their purpose.

Table 2.4 BIR Instructions

Instruction	**Description**
GOTO	Jumping to another basic block given as the operand
CALL	Invoking a method
BRANCH	Branching based on condition true or false to other basic block
RETURN	Return from current method
MOVE	Store the value to a variable at the given index location to the index specified
CONSTANT_LOAD	Load the constant value to the given index.
ADD	Add two integer values in the given indexes and store the result in the index specified
SUB	Subtract two integer values in the given indexes and store

	the result in the index specified
MUL	Multiply two integer values in the given indexes and store the result in the index specified
DIV	Divide two integer values in the given indexes and store the result in the index specified
EQUAL	Compare two values for equality and store the result as a boolean value
NOT_EQUAL	Compare two values for not equality and store the result as a boolean value
GREATER_THAN	Compare value at first index location is greater than of value at second index and store the result as a boolean value
GREATER_EQUAL	Compare value at first index location is greater than or equal of value at second index and store the result as a boolean value
LESS_THAN	Compare value at first index location is less than of value at second index and store the result as a boolean value
LESS_EQUAL	Compare value at first index location is less than or equal of value at second index and store the result as a boolean value
NEW_ARRAY	Create a new array with given type
ARRAY_ACCESS	Access the value store at index in the given array
ARRAY_STORE	Store a value at a given index location in the array
AND	Boolean AND conditional check and store the result as a boolean

OR	Boolean OR conditional check and store the result as a boolean

All the above instructions and constructs are important to understand as it will be heavily used by the JVM compiler backend to properly map the BIR instructions to JVM bytecode instructions and JVM constructs.

2.4 JVM Class File

The Java class file is a binary representation with set of bytecodes and other meta information that are used by the JVM to execute the file. This is known as the compile code of a JVM based language and since it is executed by the JVM, it is hardware and operating system independent. This makes the class file portable across any system that the JVM can run. A java class file contains the format as below.

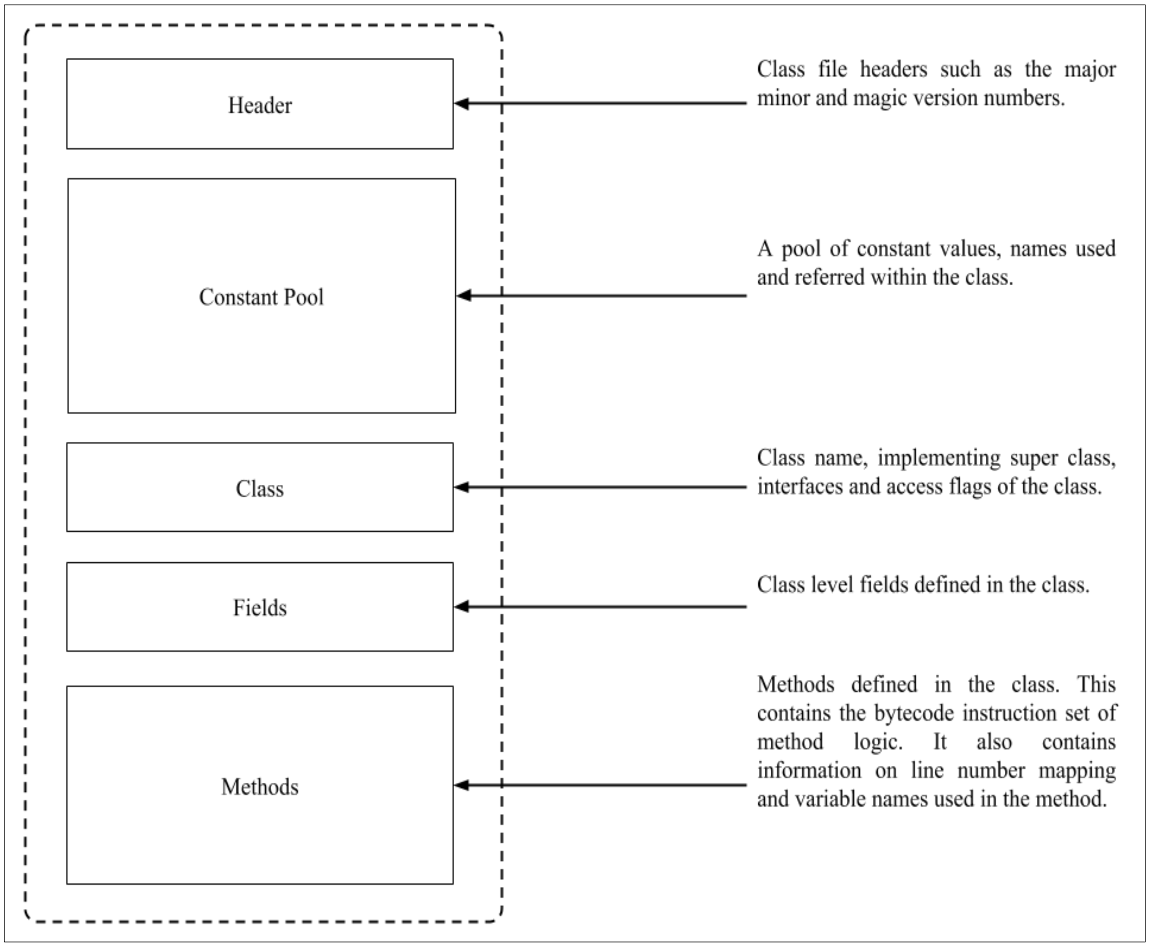

Header
Constant Pool
Class
Fields
Methods
Class file headers such as the major minor and magic version numbers.
A pool of constant values, names used and referred within the class.
Class name, implementing super class, interfaces and access flags of the class.
Class level fields defined in the class.
Methods defined in the class. This contains the bytecode instruction set of method logic. It also contains information on line number mapping and variable names used in the method.

Figure 2.5 Java Class Structure

When generating a java class file manually, the format should adhere to the above which is mandated by the JVM runtime specification. There are various helper tools available such as ASM [23] that provides ways to generate and manipulate Java bytecode generation [14]. The next thing to understand is that how JVM bytecode are executed by the JVM.

2.5 JVM Runtime Execution Model

The JVM runtime follows a stack based execution model [15]. Each JVM level thread at runtime is given a stack of frames. The frames are the execution unit which gets created for each method execution and stored in the stack. The major components found in a stack frame, which are used when a method is executed, is described in Figure 2.6.

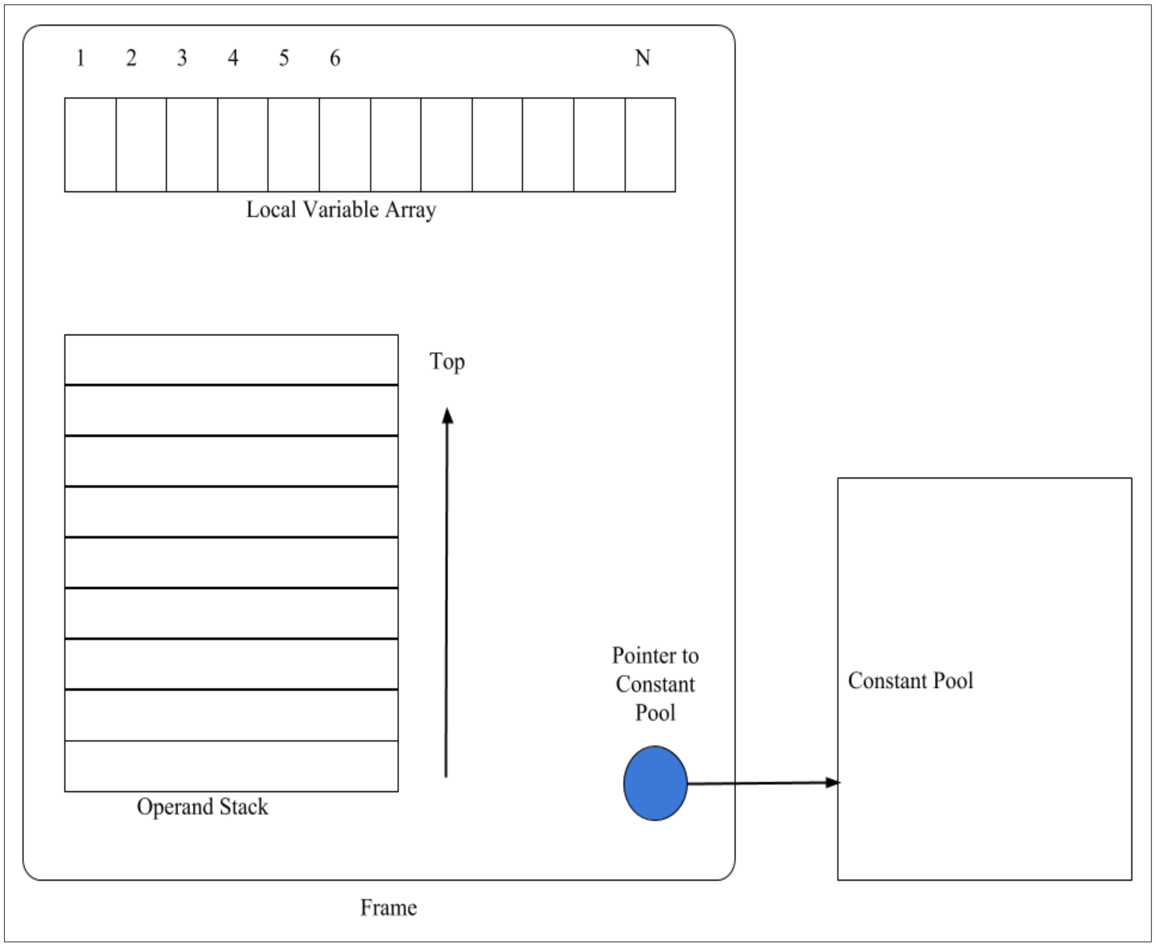


Figure 2.6 JVM Stack Frame

A frame consists of a local variable array and an operand stack. Additionally, it also holds a pointer to the constant pool from the class file. For each method invocation, a new frame is created by the JVM and when the method execution is completed, the stack frame will be destroyed. The maximum size of the local variable array and the operand stack will be calculated at compile time by the Java compiler. However, if the source language is not java, then the size of the local variable array and the operand stack has to be calculated by the source language compiler which produces the target JVM bytecode. At any point, there will be only one stack frame executed by the JVM, which is known as the current stack frame. If the execution of the current frame is passed to another method before completion, then the current stack frame will be put onto the JVM stack which holds all the frames. When a new method is invoked from current method, then a new frame gets created and it becomes the current active frame in execution. Likewise, the frames gets created and destroyed during the execution of methods.

The local variable array of each frame is used to store the values that are used within the frame execution scope. The array is ordered according to the index which starts from zero. The size of each element in the array is the size of java integer. Therefore JVM types such as int, boolean, byte, char, float can be stored at a single location in the array. However for types such as long and double, two consecutive array locations will be needed as their size exceeds the integer size. For these types long and double, they are accessed using the lower index value of the two consecutive index. For other types, since they will occupy only one array location, they are access using the same index location that they were assigned. The arguments of a method, or the values of the argument are typically stored in the initial location of this local variable array. For example, if the method access two arguments of type long and boolean, then the array index "0" will be used to get the value of the first argument of type "long" and array index "2" will be used to get the second argument which is of type boolean. We can note that we have skip the index by one, to get the second argument here because of the first argument type being "long". This information is needed when implementing the JVM compiler backend as possible the Ballerina "int" type will be represented using JVM "long" type according to the specification [17]. Another point to note that, if the method invocation is part of an object instance invocation, then the first index location of this array will typically contain the reference to the object instance itself.

The operand stack of a stack frame is used for loading the operand values of instruction set. The values can be constant values or values of variables that are stored in the local variable array. When instructions are executed by the JVM, the required operands or rather the values for the instruction will be loaded to this operand stack and the instruction will use them to perform the execution. Once operating on the operands, the result will be pushed back onto the stack. For example, the integer multiply instruction “imul” pops two values at the top of the stack, perform the multiplication and will push the result to the top of the stack. Therefore before this instruction is executed, the two values has to be pushed onto that stack. If resultant the value of an instruction needs to be used, then it has to be stored back to the local variable array.

2.6 JVM Instruction Set

The JVM instructions [17] are called bytecode as they are of size one byte. Since one byte is used to represent all the bytecodes, there can be maximum of 255 instructions at JVM level. Each instruction contains an opcode and optionally, set of operands. These operands are used by the opcode operation to load or store values from/to local variable array form/to operand stack. However most of the instructions at JVM level has no operands and they directly operate on values loaded onto the stack. For readability aspect, the instruction set are give mnemonics based representation. For example, “iload” instruction loads an integer value on to the stack. The mnemonic letter “i” describes that this instruction operate on type integer. Likewise, there are meaningful letters based on types, are given to most of the instruction on understanding its operations. If the type mnemonic is not given, then those instructions operate as common instruction on any type.

The instruction set of JVM can be divided based on their operation types. The most common operation are the load and store instructions which load values from local variable array and store the values from operand stack back to the local array. Some load instructions load values from local variables and some load values as constants. Arithmetic related instructions such as “iadd”, “isub”, “idiv”, “iand” etc operate on typically two operand values and push the result to the top of the stack. The comparison instructions such as “lcmp”, “fcmp”, “dcmp”, etc compare the values on to top of the stack for specific operation type, such as less than, greater than and pushes the result as a boolean flag to the top of the stack. Type conversion instructions such as “i2l”, “f2l”, etc operate on either narrowing the type and broadening the type. The value will be converted to the target type which can result in information being lost, if it is a narrowing conversion or information is not lost if it is the case of widening conversion. Object creation and manipulation related instructions such as “new”, “newarray” operate on creating class instances of object or arrays.

Branching and conditional related operations such as “ifeq”, “ifne”, etc compares the value on top of stack for a specific value, such as 0 and branches to the target instruction code. Similarly, comparison operations such as “if_icmplt”, “if_icmpgt”, “if_acmpeq” etc compares the two values on top of the stack, for condition such as less than, greater than, and branch to the target instruction pointed by the instruction pointer operand. Method invocation related instructions are special type of instruction which the JVM does another method invocation. For instructions such as “invokevirtual”, “invokespecial”, the JVM does an invocation in the instance level methods. The “invokestatic” instruction invokes the static method in a class based on the given name and method signature. The method return instructions such as “lreturn”, “areturn” will return from the execution of a method with the return value being on the top of stack.

These instruction and how they operate are important information for JVM compiler backend target as the correct instruction or collection of instructions has to be used when mapping the same with JVM compiler backend for ballerina.

3. METHODOLOGY

The Ballerina Intermediate Representation is considered low level compared to the ballerina source. As explained in the compiler architecture, there can be multiple backend for the same BIR representation. However considering the advantages that JVM runtime will offer, such as, inbuilt garbage collection, support for various high performing libraries and frameworks such as NettyIO based HTTP framework, Just-In-Time compilation of the Java bytecode. etc, mapping BIR to Java Intermediate Representation would be the ideal approach to follow. This will be known as the JVM based compiler backend for BIR.

3.1 JVM Compiler Backend For Ballerina

The JVM compiler backend would be responsible for generating the JVM bytecode for ballerina source. This JVM backend can be completely written in ballerina itself therefore it becomes easy when effort to write the ballerina compiler in ballerina itself. There will be many components that will be in the JVM backend. Figure 3.1 describes these components and how they will work together as a compiler backend.

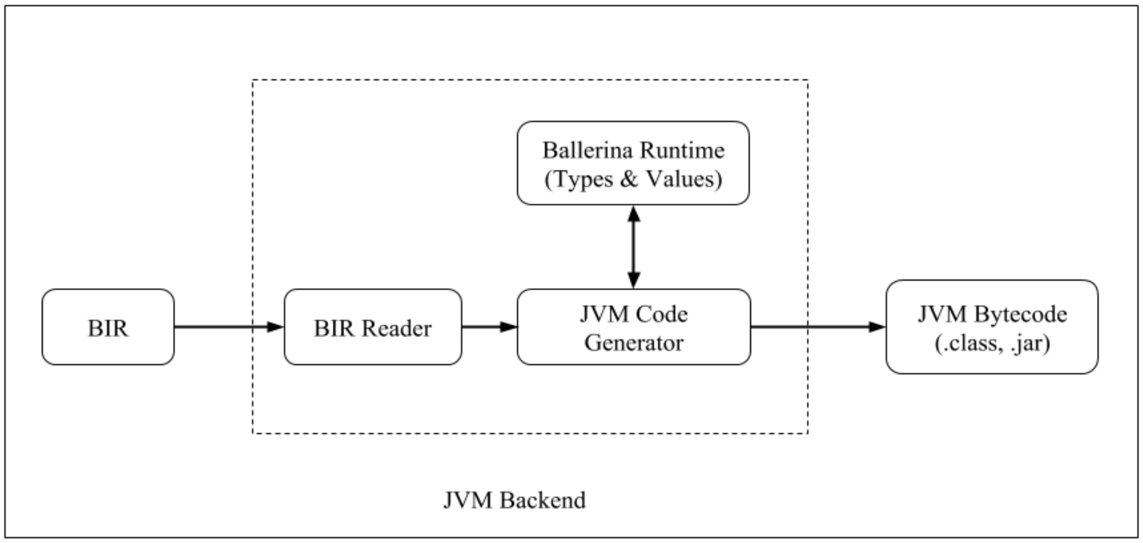


Figure 3.1 Proposed JVM Compiler Backend For Ballerina

The BIR will be read by the BIR reader which will parse the BIR graph and generate the model representation of it. The BIR model will be then fed into the JVM code generator which will map the BIR constructs (packages, variables, basic blocks, instructions) to the JVM bytecode representation. When doing the mapping, the code generator tool will use the ballerina runtime library that will hold the runtime mapping of all types and values.

There will be three major phase in compiling the ballerina source to Java bytecode. First the source has to be converted into an AST. The AST should have all the information that the next phase would need. The next phase would convert the given AST to a BIR. This is where the AST will get lowered to even low level of instruction set. The BIR should capture all the necessary information for the next phase to use. The BIR will then be converted to the target java byte code. This is the phase where the java runtime implementation should be linked with the BIR model therefore the correct execution code is generated for the BIR.

3.2 AST to BIR Generation

Ballerina compiler transforms the source into AST which is a tree based representation of the source. However BIR is considered more low level than of AST and it will contain low level details about the program. As explained previously, the BIR contains of functions, variables, operations in the form of BIR instruction which are grouped together by basic blocks. This phase could be considered as an extended phases of the compiler frontend. Since currently the compiler frontend is written in java, this phase of AST to BIR generation also is written in java. Using the AST tree node structure, the AST model will be mapped to the BIR model.

For each of the functions in AST, the required variables (local, arguments, return) will be calculated and defined in the BIR and then the function body of each function will be processed and the relevant instructions and basic blocks will be generated. And finally, the generated BIR will be written as a binary file which will mark the end of this phase.

3.3 BIR to JVM Target Generation

BIR is an intermediate form of the ballerina source however it is more closer to the low level machine code. It is platform/runtime independent and contains all the required data to generate a target executable backend.

As explained in the previous section, The BIR will be written to its binary form after the AST to BIR generation phase. The binary file will consists the BIR model for a given ballerina source file. This binary file can now be read and the further converted to the target executable code. The reading and converting of BIR is written completely in ballerina itself with aim to compile everything to a target executable code. This is needed for the bootstrapping process of converting ballerina compiler and runtime which can be written in ballerina itself. When we read the BIR binary content and then populate the BIR model in ballerina, the same model or the Java model that was used in the AST to BIR generation phase needs to be created at this phase as well. However the most important task with JVM target is how to model the ballerina type system. Once the type system is modeled, then the ballerina project and the constructs such as modules, sub-modules, imported modules, functions, basic blocks, basic block instructions has to be modeled and mapped to Java level constructs. One these constructs are modeled, the build command has to be updated on how an executable java archive and classes needs to be generated. The next section describes about the implementation details on how each of the ballerina and BIR constructs are mapped to java and how it was implemented.

4. IMPLEMENTATION

The implementation section explains about how BIR constructs were modeled in JVM level and how each instruction from BIR was mapped to JVM level instructions in generating the bytecode.

4.1 Modeling Ballerina Types & Values

The most important task in generating the target java based bytecode using JVM compiler target is how to model the ballerina type system at JVM level. This includes both types and values from ballerina type system and how it should be mapped to JVM types and values. The various types in ballerina type system and how they were mapped to JVM type system is described in the following section in detail.

4.1.1 Simple Basic Types

The types \`int\`, \`float\`, \`string\`, \`boolean\`, \`byte\` and \`nil\` are called simple basic types. Mapping these types to Java is given in Table 4.1.

Table 4.1 Ballerina Basic Value Type Mapping

Ballerina Basic Type	**Java Type**
int	long
float	double
string	java.lang.String
boolean	boolean
byte	byte
nil	null

4.1.2 Structured & Behavioral Types

The types `map`, `arrays`, `records` are called structural types and `objects` is a behavioral type. Table 4.2 presents the proposed approach to take when modeling structured & behavioral type with JVM target. This modeling will be done as the future addition to this project.

Table 4.2 Ballerina Structured & Behavioral Type Matching

Structured & Behavioral Types	**Java Type Model**
map	BallerinaMap (which could extend from HashMap to support ballerin level inbuilt functions)
arrays	For each basic types, we can have a array representation (int[] values -> BIntArray(long[] values)). And for value based arrays, we can have a general BValueArray representation (Foo[] values -> BValueArray(BValue[] values)).
records	Can extend from BMap. However as an optimization on closed records, we can code generate the Java class representation of the record, which could extend form BValue, where each filed becoming class instance variables.
objects	Can be represented using Java class and it where object fields becoming java class instance variables.
object functions	Each function can be modeled as a separate class which has access to it enclosing object instance. A function class representation can implement from an interface

	(BFunction) that has a single method called invoke(). The ballerina function arguments can be class level variables and the return value also can be class level variables.

4.2 Modeling Ballerina Project

Once the types and values are modeled, next task is to model the ballerina projects and its structure at JVM level. A JVM package consists of one or more JVM class files. A JVM class file needs a class name and a constructor method, at minimum, in order for it to be used by other class and invoked. Once we have the class name and the initializing method, the rest of the work is to generate the other class level methods and variables.

However the most important difference between BIR model and JVM model is that BIR is a register based approach for variables load and store in method local registers, where as JVM is a stack based machine where variable are load and store in method local stack known as the operand stack. Therefore the important step in mapping BIR instructions is how do we map the variable and their index which is of register based approach to variables and index which is of stack based approach.

When mapping the BIR model to JVM class files, the below steps were followed.

1. Modeling package
2. Modeling the class with the description
3. Generation of method(s) description and signatures
4. Generation of method(s) body
5. Generation of class file content into binary a file (.class)

Following section provides detail on each of the majors steps in modeling the BIR to java packages.

4.2.1 Modeling package

Ballerina package system follows a very similar approach on java packages. A module consists of one or more ballerina source files which can be under various sub directories in a module. For example, the ballerina/http module consists a directory structure as given in Figure 4.1.

```
|____ballerina
| |____http
| | |____websocket
| | |____auth
| | |____http2
| | |____resiliency
| | |____redirect
|_|_|____caching
```

Figure 4.1 Directory Structure of ballerina/http Module

In Figure 4.1, we can see that there are different subdirectories under http where each contain the related .bal source files. The approach we can follow in converting the module structure to a java packages is, treating the root organization name (ballerina/http in this case) as the root package at java and treat the rest of the sub directories as sub packages in java level.

```
|____src
| |____main
| | |____java
| | | |____ballerina
| | | | |____http
| | | | | |____websocket
| | | | | |____auth
| | | | | |____http2
| | | | | |____resiliency
| | | | | |____redirect
|_|_|_|_|_|____caching
```

Figure 4.2 Java Package Structure of ballerina/http Module

After defining the package structure, we can follow the approach of creating separate class files for each of the ballerina source file (.bal). The ballerina source file name can be used for the class file name. Therefore when we convert the ballerina/http module in to a java project, the generated packages structure will look like described in Figure 4.2

4.2.2 Modeling the class with the description

The ballerina source file name was used as the class name and also as the binary class file name as well. This phase of the implementation uses an approach to create a default constructor for the class which only initializes the class instance, whereas a future implementation could create a constructor based on some further processing of the ballerina source file and create a constructor or multiple constructor with initializing class level variables and their default values, etc. For example, if the ballerina source file name is *"httpRequest.bal"*, then a class file with the name *"httpRequest.class"* will be created with a default constructor. Each functions is a ballerina source can be mapped to Java class level methods. Table 4.3 shows how a ballerina source file is mapped to BIR and then how it will be mapped to a Java class file.

Table 4.3 Example Ballerina Source To Java Class Mapping

Example in Ballerina	**Example in BIR**

function foo (int a) returns int {
    int b = a + 10;
    return b;
}
	function foo(int) -> int {
	        int %0; // return
	        int %1; // arg
	        int %2; // local
	        int %3; // temp
	        int %4; // temp
	        int %5; // temp
	        int %6; // temp
	
	        bb0 {
	                %3 = %1;
	                %4 = const_load 10;
	                %5 = add %3 %4;
	                %2 = %5;
	                %6 = %2;

	%0 = %6;
	goto bb1;
	}
	
	bb1 {
	return;
	}
	}

Example in Ballerina (test.bal)	**Example in generated Java class**

function foo (int a) returns int {
int b = a + 10;
return b;
}
	public class test {
	public test() {
	}
	
	static long foo(long var0) {
	long var6 = 10L;
	long var8 = var0 + var6;
	return var8;
	}
	}

4.2.3 Generation of method(s) description and signatures

A Java method signature contains the following details

1. Access flags of the method
2. Name of the method
3. Description of the method
4. Method signature
5. Exceptions that the method would throw.

Ballerina functions has two access modifiers, module private and public. As we are mapping ballerina modules to java package, the ballerina package level access modifier can be mapped. And the public function can be public methods in the java class. Since there can be only instance of the class for a ballerina file, the approach is to generate static methods for each function in a ballerina source file. Figure 3.2 shows how a ballerina function with the given signature gets mapped and generated to java level method with given method signature.

```
public function sort(int[] arr, int low, int high) returns int[] {
    // function body
}
```

```
public static long[] sort(long[] var0, long var1, long var3) {
    // method body
}
```

Figure 4.3 Java Method Signature Mapping

Using the Java bytecode representation, the function described in Figure 4.3 will have a Java level method generated given in Figure 4.4.

```
public static long[] sort(long[], long, long);
  descriptor: ([JJJ)[J;
  flags: ACC_PUBLIC, ACC_STATIC
```

Figure 4.4 Java Method Signature Example

In here, the signature “J” is used by Java bytecode to represent `long`. Likewise, the Java method signature can be generated using the ballerina type and value mapping and model which is defined.

4.2.4 Generation of method body

This is the most important step as it involves generating the method body. This step can be further broken down as below.

1. Processing of method return value
2. Processing of method arguments
3. Processing of method basic blocks
4. Processing of basic block instructions

5. Processing of basic block termination

4.2.5 Processing of method return value

Let’s take the BIR example as given in Figure 4.5. This function takes a integer argument and returns integer. The BIR model represents the return index as the first variable in its list of method local variable. However for Java bytecode, the return index is only needed at the end of method processing and the value to be returned should be loaded to the stack as the top element. Therefore when mapping the BIR return var index to Java bytecode, we have to calculate the index at the beginning of the BIR function body processing as it is the first element. And at the end of the function processing, this index of the return variable should be used to load the actual value of the return variable onto the stack.

```
function foo(int) -> int {
        int %0;         // return
        int %1;         // arg
        int %2;         // local
        int %3;         // temp
        int %4;         // temp
        int %5;         // temp
        int %6;         // temp

        bb0 {
                %3 = %1;
                %4 = const_load 10;
                %5 = add %3 %4;
                %2 = %5;
                %6 = %2;
                %0 = %6;
                goto bb1;
        }

        bb1 {
                return;
        }
}
```

Figure 4.5 Example BIR

4.2.6 Processing of method arguments

Once we have processed the return value, the rest of the local variable in the method body represents the method arguments and the method local variables used. When mapping integer or float type arguments to java long or double, we have to use two consecutive index [17] values as per the JVM bytecode specification. For the rest of the types, we can use a single index value to map with argument index value.

Mapping BIR variable declaration to JVM index value is is represented using the ballerina object given in Figure 3.4, which models a mapping relationship with BIR variable declaration to JVM index.

```
type BIRVarToJVMIndexMap object {
    private int localVarIndex;
    private map<int> jvmLocalVarIndexMap;

    function add(bir:VariableDcl varDcl) {
        string varRefName = getVarRefName(varDcl);
        jvmLocalVarIndexMap[varRefName] = localVarIndex;
        match varDcl.typeValue {
            bir:BTypeInt => {
                localVarIndex = localVarIndex + 2;
            }
            any => {
                localVarIndex = localVarIndex + 1;
            }
        }
    }

    function getIndex(bir:VariableDcl varDcl) returns int? {
        string varRefName = getVarRefName(varDcl);
        if (!(jvmLocalVarIndexMap.hasKey(varRefName))) {
            return -1;
        }

        return jvmLocalVarIndexMap[varRefName];
    }
};
```

Figure 4.6 BIR Variable To JVM Index Mapping

This mapping object internally keep a ballerina map of integer values. Whenever a new variable declaration is added to the map, it checks whether the type of its `int` and increment the current variable index by two, else, for any other type, it increments by one. Once the variable is added to the map along with its calculated index, it can be later required to get the index value. Most importantly, there is a new instance of this map for each of the function is the BIR model as each method in the JVM will have its separate invocation stack with index values.

4.2.7 Processing of method basic blocks

A basic block in BIR consists a group of executable statements. The end of a basic block is either a branch or goto statement to other basic blocks or return/termination statement. Therefore as a general model to handle goto in JVM side, we can create separate JVM labels for each basic block therefore the branching can be handled for one basic block to another.

Table 4.4 describes an example ballerina program which has conditional branching using the if-else statement.

Table 4.4 Basic Block Generation

Ballerina source	**BIR**

```
function foo(int a) returns int {
   int b = 5;
   if (a < 4) {
       b = 7;
   } else {
       b = 9;
   }
   return b;
}
```
	```
	function foo(int) -> int {
	        int %0;        // return
	        int %1;        // arg
	        int %2;        // local
	        int %3;        // temp
	        int %4;        // temp
	        int %5;        // temp
	        boolean %6;            // temp
	        int %7;        // temp
	        int %8;        // temp
	        int %9;        // temp

	        bb0 {
	```

```
        %3 = const_load 5;
        %2 = %3;
        %4 = %1;
        %5 = const_load 4;
        %6 = less_than %4 %5;
        branch %6 [true:bb1, false:bb2];
    }

    bb1 {
        %7 = const_load 7;
        %2 = %7;
        goto bb3;
    }

    bb2 {
        %8 = const_load 9;
        %2 = %8;
        goto bb3;
    }

    bb3 {
        %9 = %2;
        %0 = %9;
        goto bb4;
    }

    bb4 {
        return;
    }
}
```

There are five basic blocks created for this function. Even though this BIR model is not optimized (as there are lot of unwanted statements which could have been removed), it has all the required statements for generating the target executable code. Therefore when modeling these basic blocks in JVM, we can use JVM labels for each of the basic block using the basic block id with enclosing function name as the label id. The generated JVM bytecode for above program will be like below and the highlighted section (**goto instruction**) are the labels that gets created for basic blocks.

```
public static long foo(long);
     0: ldc2_w        #11    // long 5l
     3: lstore        4
     5: lload         4
     7: lstore_2
     8: lload_0
     9: lstore        6
    11: ldc2_w        #13    // long 4l
```

```
14: lstore       8
16: lload        6
18: lload        8
20: lcmp
21: iflt        28
24: iconst_0
25: goto        29
28: iconst_1
29: istore      10
31: iload       10
33: ifgt        39
36: goto        50
39: ldc2_w       #15   // long 7l
42: lstore      11
44: lload       11
46: lstore_2
47: goto        61
50: ldc2_w       #17   // long 9l
53: lstore      13
55: lload       13
57: lstore_2
58: goto        61
61: lload_2
62: lstore      15
64: lload       15
66: lstore      17
68: goto        71
71: lload       17
73: invokestatic  #24   // Method java/lang/Long.valueOf:(J)Ljava/lang/Long;
76: areturn
```

Figure 4.7 Java Bytecode Generated

4.2.8 Processing of basic block instructions

As explained above, each basic has a group of statements or rather set of instructions, where, each of them has to be property code generated to the target executable. The set of instructions supported by BIR is already mentioned and for this project effort following instructions were explored to support the test scenarios.

4.2.9 Constant Load Instruction

Constant load deals with how to define new literal values (eg : int a = 12;). Therefore when there is a new variable definition with a literal value on the right hand side, then a constant load instruction will be generated. Let's looks at an example as below.

Table 4.5 Constant Load Instruction Generation

Ballerina source	**BIR**

function foo (int a) returns int {
    int b = 10;
    return b;
}
	function foo(int) -> int {
	        int %0; // return
	        int %1; // arg
	        int %2; // local
	        int %3; // temp
	        int %4; // temp
	
	        bb0 {
	                %3 = const_load 10;
	                %2 = %3;
	                %4 = %2;
	                %0 = %4;
	                goto bb1;
	        }
	
	        bb1 {
	                return;
	        }
	}

We can see that for the variable definition in ballerina source (int b = 10) there is a constant load instruction at BIR (%3 = const_load 10).

However when we try to generate the above in Java byte code, we have to map the constant_load instruction to a similar one in JVM. As mentioned previously, BIR is a register based machine where variables are store and loaded from method local registers. Since JVM is based on stack based architecture, we would need to generated the index and store the constant values. For example, LSTORE instruction in JVM requires the index on where to store value from the current operand stack.

Therefore when we consider a single constant_load instruction for BIR as below, when we map to JVM instruction, we need two instruction due to nature of stack based operations on JVM.

This instruction “%3 = const_load 10” has a load and a store operations. Therefore on JVM we have first load to literal value 10 onto the stack, and then store that from stack to the local variable array. The loading part will use the value from Constant Load instructions and storing part will get the index calculated for the variable declaration (int a) and store it. Therefore the final generation would be as below.

Table 4.6 Constant Load Instruction Mapping

Ballerina	**BIR**	**JVM**
int b = 10;	%3 = const_load 10;	ldc2_w #10 // long 10l lstore 4

Likewise, for different types, we can use the similar approach on loading and storing the literal values based on their load and store instructions mapping on JVM. For example literal String values in Java are considered as general object references and they are modeled as ALOAD and ASTORE instructions.

4.2.10 Move Instruction

A move instruction in BIR is an assignment statement where the value of reference variable on right hand side of the assignment is assigned as the value of the reference variable on left hand side. Therefore it can be modeled as a load and store values based on the index of rhs and lhs variables. For example below ballerina code has a variable definition and then followed by a variable assignment. In BIR it is modeled as a cons_load instruction followed by a move instruction. And on JVM side, we will have load and a store instruction that maps the move instruction from BIR.

Table 4.7 Move Instruction Mapping

Ballerina	**BIR**	**JVM**

int b = 10;
int c = b;
	%3 = const_load 10;
	%2 = %3;
		ldc2_w #10 // long 10l
		lstore 4
		lload 4
		lstore 6

Similarly for other types supported, we can generate the JVM load and store instructions as below.

Table 4.8 Types of Move Instruction Mappings

Ballerina Type	**JVM Instruction Type**
int	LLOAD, LSTORE
boolean	ILOAD, ISTORE
string, arrays	ALOAD, ASTORE

4.2.11 Binary Operation Instructions

Binary operations are one of commonly used operation in a ballerina source file. There are various binary operations that are supported in ballerina and BIR. We can look at each of them on how to map them to JVM instructions.

4.2.12 Add Instruction

An add instruction basically works on adding two operands which are mostly of the same type and produce the result. For example, and add instruction for integer type will add do a arithmetic addition on the integer values of the two operands and provide the result.

Whereas the add instruction on two string values will concatenate the two string and produce the end result. Therefore based on the type of the variable the add instruction has to be generated at JVM level.

Table 4.9 Add Instruction Generation

Ballerina	**BIR**

function foo(int a, int b) returns int {
   int c = a + b;
   return c;
}
	function foo(int,int) -> int {
	    int %0; // return
	    int %1; // arg
	    int %2; // arg
	    int %3; // local
	    int %4; // temp
	    int %5; // temp
	    int %6; // temp
	    int %7; // temp
	
	    bb0 {
	        %4 = %1;
	        %5 = %2;
	        %6 = add %4 %5;
	        %3 = %6;
	        %7 = %3;
	        %0 = %7;
	        goto bb1;
	    }
	
	    bb1 {
	        return;
	    }
	}

In Table 4.9, the BIR instruction "%6 = add %4 %5" has two operands and a store value. It takes the operands %4 and %5 and add them and store the value to another variable. However when we are modeling the same with JVM, we need to first load both operands to stack, perform and add and then store the result at the target index. Therefore it will contain four (04) JVM instructions as below.

Table 4.10 Add Instruction Mapping

Ballerina	**BIR**	**JVM**
int c = a + b;	%6 = add %4 %5;	lload 6 lload 8 ladd

		lstore 10

For string type with add instruction, it has to be string concatenation at JVM level. Therefore we have to generate JVM instruction set as below where the "*invokevirtual*" method is the invocation of "*String.concat"* system method at JVM.

Table 4.11 String Concatenation Mapping

BIR	**JVM**
%6 = add %4 %5;	aload 6 aload 8 invokevirtual #24 astore 10

Similarly, based on the type from BIR, we can generate the correct set of "Add" instructions at JVM level.

4.2.13 Subtract Instruction

This is similar to the add instruction above and we have to generate the JVM instruction that handles the subtraction as below.

Table 4.12 Subtract Instruction Mapping

Ballerina	**BIR**	**JVM**
int c = a - b;	%6 = sub %4 %5;	lload 6 lload 8 **lsub** lstore 10

Even though we have add symbol supported for string types, which effectively does string concatenation, there is no such support with subtract operation.

4.2.14 Multiply & Divide Instructions

Table 4.13 explains the JVM bytecode generation for multiply and divide related instructions.

Table 4.13 Multiply & Divide Load Instruction Mapping

Ballerina	**BIR**	**JVM**
int c = a * b;	%6 = mul %4 %5;	lload 6 lload 8 **lmul** lstore 10
int c = a / b;	%6 = div %4 %5;	lload 6 lload 8 **ldiv** lstore 10

4.2.15 Equal Instruction

Then equal operation checks for value equality and produce a boolean result which can be used in conditional branching using if statements. A simple example and the generated BIR would be as below.

Table 4.14 Equal Instruction Generation

Ballerina	**BIR**

if (a == b) {
    return 23;
}
return 14;
	%5 = equal %3 %4;
	branch %9 [true:bb1, false:bb2];

1. Create two labels (lable1, lable2) that handles the true and false conditions of the **Equal** operation.
2. Load the value of first operand onto the stack.

3. Load the boolean value of second operand onto the stack.
4. Compare both and if not equal (i.e the **false** case), then goto label1.
5. Or else, go to label2, which will be the **true** case.
6. When visiting label1, it will store integer value 0 (false) as the result of the AND operation.
7. When visiting label2, it will store integer value 1 (true) as the result of the AND operation.

Table 4.15 Equal Instruction Mapping

BIR	**JVM**

%9 = equal %5 %8;
branch %9 [true:bb1, false:bb2];
	LLOAD
	LLOAD
	LCMP
	IFNE L1
	ICONST_1
	GOTO L2
	L1
	ICONST_0
	L2
	ISTORE

4.2.16 Condition Based Instructions

The condition based instruction that BIR currently supports are Less Than, Less Than or Equal, Greater Than, Greater Than or Equal. The difference among these operations is the conditional check on values only. Other instruction will be same as with equal instruction. Table 4.16 explains the bytecode instructions and their different types for each of these operations.

Table 4.16 Condition Based Instruction Mapping

Type	**BIR**	**JVM**
less_than	%9 = **less_than** %5 %8; branch %9 [true:bb1, false:bb2];	LLOAD LLOAD LCMP **IFLT L1** ICONST_1 GOTO L2 L1 ICONST_0 L2 ISTORE
less_than_equal	%9 = **less_equal** %5 %8; branch %9 [true:bb1, false:bb2];	LLOAD LLOAD LCMP **IFLE L1** ICONST_1 GOTO L2 L1 ICONST_0 L2 ISTORE
greater_than	%9 = **greater_than** %5 %8; branch %9 [true:bb1, false:bb2];	LLOAD LLOAD LCMP **IFGT L1** ICONST_1 GOTO L2 L1 ICONST_0 L2 ISTORE
greater_equal	%9 = **greater_equal** %5 %8; branch %9 [true:bb1, false:bb2];	LLOAD LLOAD LCMP **IFGE L1** ICONST_1 GOTO L2 L1 ICONST_0 L2 ISTORE

4.2.17 AND Instruction

The AND and OR operations will generate few more than instructions to handle the conditions and branching related logic at BIR level as below.

Table 4.17 Binary AND Instruction Generation

Ballerina	**BIR**

if (a == 1 && b == 3) {
    return 23;
}
return 14;
	%4 = const_load 1;
	%5 = equal %3 %4;
	%7 = const_load 3;
	%8 = equal %6 %7;
	%9 = and %5 %8;
	branch %9 [true:bb1, false:bb2];

For both RHS and LHS expressions with the AND symbol, separate instructions set will be created at BIR side as above. In AND, both operands (RHS, LHS) are of boolean type. Now when we map this to the JVM bytecode level, we have to create labels and jump/goto instructions based on the conditions. The logic to generate JVM bytecode set would be as below.

1. Create two labels (lable1, lable2) that handles the true and false conditions of the AND operation.
2. Load the boolean value (can be 1 or 0) of first operand onto the stack
3. Load integer value 1 onto the stack.
4. Compare both and if not equal (i.e the **false** case), then goto label1.
5. Similarly, load the boolean value of second operand onto the stack.
6. Load integer value 1 onto the stack.
7. Compare both and if not equal (i.e the **false** case), then goto label1.
8. Or else, go to label2, which will be the **true** case.
9. When visiting label1, it will store integer value 0 (false) as the result of the AND operation.
10. When visiting label2, it will store integer value 1 (true) as the result of the AND operation.

Table 4.18 Binary AND Instruction Mapping

BIR	JVM

%9 = and %5 %8;
branch %9 [true:bb1, false:bb2];
	ILOAD
	ICONST_1
	IF_ICMPNE L1
	ILOAD
	ICONST_1
	IF_ICMPNE L1
	ICONST_1
	GOTO L2
	L1
	ICONST_0
	L2
	ISTORE

4.2.18 OR Instruction

For OR operation, the difference would the boolean condition check on both RHS and LHS as it is required at least one of them evaluating to true. The relevant JVM instruction set that will generated is given in Table 4.19.

Table 4.19 Binary OR Instruction Mapping

BIR	JVM

%9 = or %5 %8;
branch %9 [true:bb1, false:bb2];
	ILOAD
	ICONST_1
	IF_ICMPEQ L1
	ILOAD
	ICONST_1
	IF_ICMPEQ L1
	ICONST_0
	GOTO L2
	L1
	ICONST_1
	L2
	ISTORE

4.2.19 Array Load Instruction

Arrays in JVM is considered as objects therefore loading array into the stack will use the same instruction as loading other reference object instances. And then accessing values from arrays will use its relevant type related instruction. For example, when we are accessing values from integer type array in ballerina, the same can be modeled as loading values from long type array in JVM.

Table 4.20 Array Load Instruction Mapping

Ballerina	**BIR**	**JVM**

int[] a = [1, 2, 3];
int b = a[2];
	%5 = const_load 2;
	%3 = array_access %4[%5]
		aload
		lload
		l2i
		laload
		lstore

4.2.20 Array Store Instruction

When a value is stored in an array on a specified index, the JVM array store bytecode for the relevant type can be generated. Below example is for string values in a long array. For reference arrays, the instruction AASTORE can be used.

Table 4.21 Array Store Instruction Mapping

Ballerina	**BIR**	**JVM**

int[] a = [1, 2, 3];
a[2] = 45;
	%2 = const_load 45;
	%4 = const_load 3;
	%3[%4] = %2
		aload
		ldc2_w #13 //long 3
		l2i
		ldc2_w #11 //long 45
		lastore

4.2.21 Loops

The looping construct in BIR currently support while loops, however when the BIR instruction is generated, we can see that, the looping is generated as set of branching and "goto" instructions with basic blocks that handles the looping construct. This is explained using the below example.

Table 4.22 Mapping of Loops

Ballerina	**BIR**

function foo(int[] array) returns int {
    int i = 0;
    int sum = 0;
    while (i < lengthof array) {
        sum += array[i];
        i++;
    }
    return sum;
}
	bb0 {
	    %3 = const_load 0;
	    %2 = %3;
	    %5 = const_load 0;
	    %4 = %5;
	    goto bb1;
	}
	
	bb1 {
	    %6 = %2;
	    %8 = %1;
	    %7 = length %8;
	    %9 = less_than %6 %7;
	    branch %9 [true:bb2, false:bb3];
	}
	
	bb2 {
	    %10 = %4;
	    %12 = %1;
	    %13 = %2;
	    %11 = array_access %12[%13]
	    %14 = add %10 %11;
	    %4 = %14;
	    %15 = %2;
	    %16 = const_load 1;
	    %17 = add %15 %16;
	    %2 = %17;
	    goto bb1;
	}
	
	bb3 {
	    %18 = %4;
	    %0 = %18;
	    goto bb4;
	}

	bb4 {
	        return;
	}

With basic blocks and termination operations on branching and traversing basic blocks, the while loops are modeled as above in BIR. The basic block termination that handles the looping is highlighted above where if the condition on **basicblock1** fails, it skips the loop and go to the **basicblock3** that return, otherwise, the execution will go the **basicblock2** which executes the while loop body. We can see from the **basicblock2** last instruction, it again goes to **basicblock1**, which starts the next iteration in the loop. Therefore since we have already modeled basicblock goto instruction, the JVM bytecode generation will work seamlessly without any need for additional instruction needed.

4.2.22 Length Instruction

We can explain how length expression is modeled the below example loops which contains array access and array length operations with an “if” conditional statement.

Table 4.23 Length Instruction Generation

Ballerina	**BIR**

function foo(int[] a, int b) {
    int c = 0;
    if (b < lengthof a) {
        c = 11;
    }
}
	%6 = length %7;
	%8 = less_than %5 %6;
	branch %8 [true:bb1, false:bb2];

Generated JVM bytecode is given in Table 4.24.

Table 4.24 Length Instruction Mapping

BIR	JVM

%6 = length %7;
%8 = less_than %5 %6;
branch %8 [true:bb1, false:bb2];
	lload 7
	aload 11
	arraylength
	i2l
	lcmp
	iflt // true label
	iconst_0
	goto // false label

The important point here is that the highlighted instructions where the “arraylength” instruction in JVM produces a JVM integer value however as we are comparing the value with a ballerina integer (which is a JVM long type), we have to do a integer to long conversion (using the “i2l” instruction).

4.2.23 Processing of basic block termination instructions

A basic block can terminate using one of the instruction such as **GOTO, Branch, Return, MethodCall.** In this, GOTO and Branch works in similar manner however with small difference on “branch” has to evaluate the true or false condition and then goto a specific basicblock, where as the GOTO directly goes to another basicblock specified.

Table 4.25 Basicblock Termination Types

Type	BIR	JVM
GOTO	goto bb3;	goto L3
Branch	branch %8 [true:bb1, false:bb2];	iload //value of boolean expr ifgt L1 goto L2

The “Return” instruction deals with returning from a method execution with or without a value. Therefore when mapping this to the JVM, we have first load the expected return value onto the stack and then call the JVM return instruction.

The loading instruction will be based on the type of value. For example, if it’s long, then it will be LLOAD at JVM level and for reference types such as string, arrays, it will be ALOAD at JVM level. Once the value is loaded, then we have to call the ARETURN instruction at JVM level.

The “Call” termination instruction is used when there is a method call to another method. It is explained using the below example.

Table 4.26 Call Instruction Generation

Ballerina	**BIR**

function foo() returns int {
  int b = bar(5);
  return b;
}
 	
function bar(int a) returns int {
  return a + 7;
}
	function foo() -> int {
	    int %0; // return
	    int %1; // local
	    int %2; // temp
	    int %3; // temp
	    int %4; // temp
	
	    bb0 {
	        %2 = const_load 5;
	        %3 = bar(%2) -> bb1;
	    }
	
	    bb1 {
	        %1 = %3;
	        %4 = %1;
	        %0 = %4;
	        goto bb2;
	    }
	
	    bb2 {
	        return;
	    }
	}
	
	function bar(int) -> int {
	    int %0; // return
	    int %1; // arg
	    int %2; // temp
	    int %3; // temp

	int %4; // temp
	
	bb0 {
	%2 = %1;
	%3 = const_load 7;
	%4 = add %2 %3;
	%0 = %4;
	goto bb1;
	}
	
	bb1 {
	return;
	}
	}

In this BIR, we can see that there is a method call (%3 = bar(%2) -> bb1) from one method to another. It is modeled as, invoke method bar with an argument at index %2 and then store the return on index %3 and then goto basicblock1. Therefore when this instruction is modeled at JVM, we also have to invoke the method, get the return value of it and then store it. However since we have modelled ballerina methods as static methods, we have to do a static method invocations with correct arguments and types. Following is the approach on finding the target methods signature and how we can load the required arguments.

The INVOKESTATIC instruction at JVM level requires the information on className, method name, and method signature as operands. For example, when invoking the bar method, we would need to generate the JVM instruction given below.

```
invokestatic    #13      // Method bar:(J)J;
```

In this example, the class would be the same class as the calling method, however the target methods signature (name and description) needs to be calculated. It can be done using the information available in the “Call” instruction. The “Call” instruction in BIR captures the target method name and its argument, return types, etc. Therefore when mapping them, we can use the conditional check on the types and generated the string with the correct signature values. For example, if the argument type is ballerina `int`, we can append the descriptor string with “J” character. Likewise, if the type is Ballerina `strings`, we can append the descriptor with “Ljava/lang/String;”.

Based on this approach, the generated JVM instructions is given in Table 4.27.

Table 4.27 Call Instruction Mapping

Ballerina	**BIR**	**JVM**

int b = bar(5);	%2 = const_load 5;
	%3 = bar(%2) -> bb1;
		lload
		invokestatic // Method bar:(J)J

If the argument is of type string or array, then we have to do “ALOAD” to load the reference value onto the stack.

4.2.24 Generation of class file content into binary a file (.class)

This is the final step after processing a BIR. Once the above mentioned instruction set are processed, a JVM class file with the set of JVM instructions will be generated which will still reside in the memory. Now the next step is to convert class file content into a binary file. As we are using the ASM library underneath to generate the JVM class file content, ASM provides a way give out a byte array for the class file content using the class writer instance. Once we get the java level byte array, we can use Java File API to write the content into a file with “.class” extension. With multiple classes, they can be gropued and written to a java archive with .jar extension using the JarOutputStream API from java.

4.3 Updating Ballerina Build Command

Once the implementation is done, next is to compile Ballerina programs to Java class files. The ballerina programs are compiled using the build command. However by default, the build command generate the ballerina compiled binary (.balx). Therefore we have use a flag in order to compile to Java class files. For this purpose, a new flag was introduced with the build command as follow.

```
$ ballerina build --jvm-target <ballerina-source-file>
```

Example :

```
$ ballerina build --jvm-target example.bal
```

When issuing the above command, a class file will be generated at the same location with the name of the given ballerina source file name.

5. RESULTS AND EVALUATION

The performance of current ballerina runtime against the JVM compiler backend can be compared using some CPU bound operations. Some of the algorithms such as fibonacci series, merge sort, matrix multiplications, string regular expression pattern matching, etc can be used as they mostly contain CPU bound operations. For evaluation purpose, few programs and algorithms which are use in this section were first written in Ballerina. Then, using the JVM based compiler backend implementation, those Ballerina programs were compiled directly into Java class files. The same program/algorithm logic is also written in pure Java and it was used as the baseline to compare the performance of both Ballerina and generated Java class files. Therefore there will be three runtimes that will be used with each test scenarios and there are BVM, JVM Target and Java.

For each of the test, the system with all three runtimes was let to warm up for 300 seconds with a moderate load. This is to make sure that the test scenarios are given fairness when the programs are running on BVM as well as directly on JVM. And each of the iteration was run for minimum of 100 times and then the average value of these 100 runs were calculated for each test iteration. This is to rule out any sudden performance spikes and degradation that can arise with systems. The test cases used for this study is available online [18]. The data sets used for the test scenarios are explained in each of the test case specific description. The results of each test scenario is also available online [25]. The system that was for these test case had the system configuration given in Table 5.1.

Table 5.1 Test System Configuration

Model Name	MacBook Pro
Processor Name	Intel Core i7
Processor Speed	2.9 GHz
Number of Processors	1
Total Number of Cores	4
Memory	16 GB

5.1 Fibonacci Series

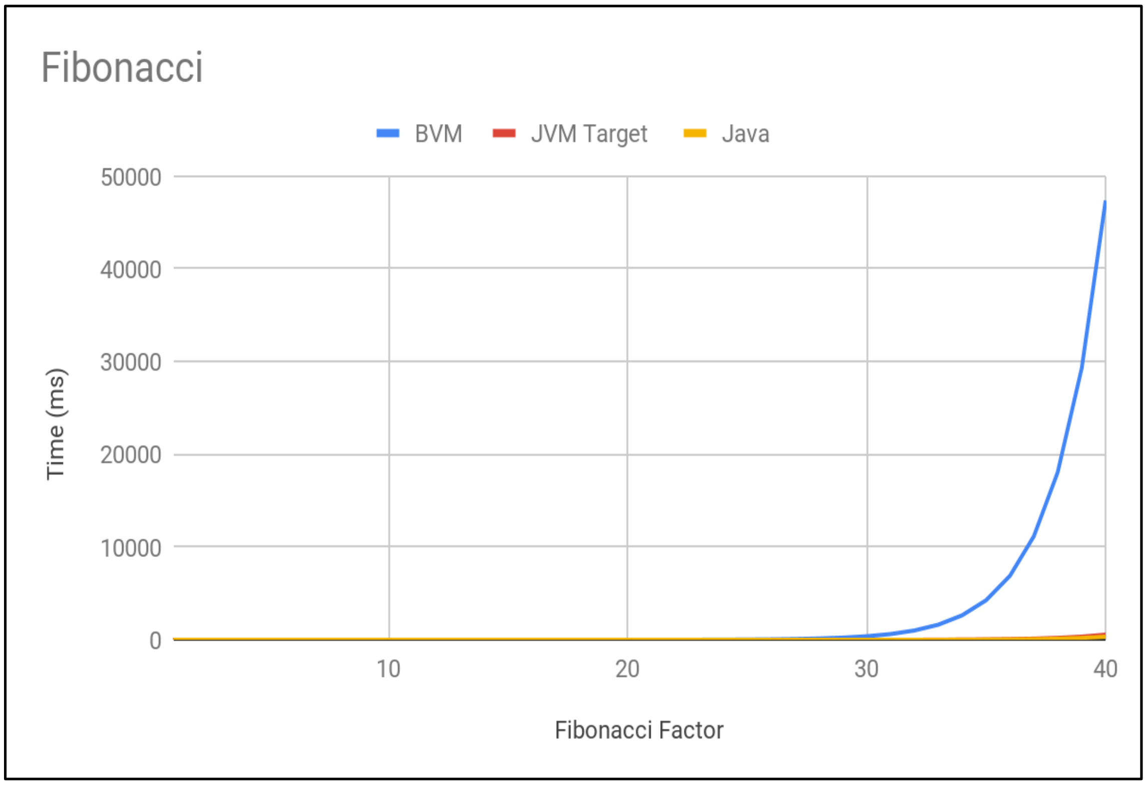


Figure 5.1 Fibonacci Test

The fibonacci series used in here [18] contains a recursive approach for calculating the value. Starting from fibonacci factor 01, this test was run up to the factor 40 and the time was measured for each factor to complete the execution. Before the test was run, the system was warmed up with adequate time. The time measured for each completions was taken as an average by running the execution for 100 times and dividing the total time by 100. This is to take the time value as an average rather than running the test just one time for each fibonacci factor. This will rule out any outlier values that may get measured during the test run.

The observation here is that the time tends to increase exponentially for BVM lot sooner than of JVM target or pure Java based programs. The time that BVM took to complete the fibonacci factor 40 is almost 100 times that time value that JVM target has took. Also, the JVM target time values were very close to the pure Java based program.

5.2 Merge Sort

The merge sort algorithm used here [18] uses a recursive approach to sort a given integer array. The algorithm first slice the integer array into two halves and then sort each half recursively. Then the sorted halves are merged together into a single array. This programs was warmed up with adequate time before the time values were calculated. The test input array size starts from 1000 and it was increased up to 1000000 with a predefined interval. The above graph is drawn using log scale for the x axis, which is the array size, to show the results in a more meaningful manner with array size interval.

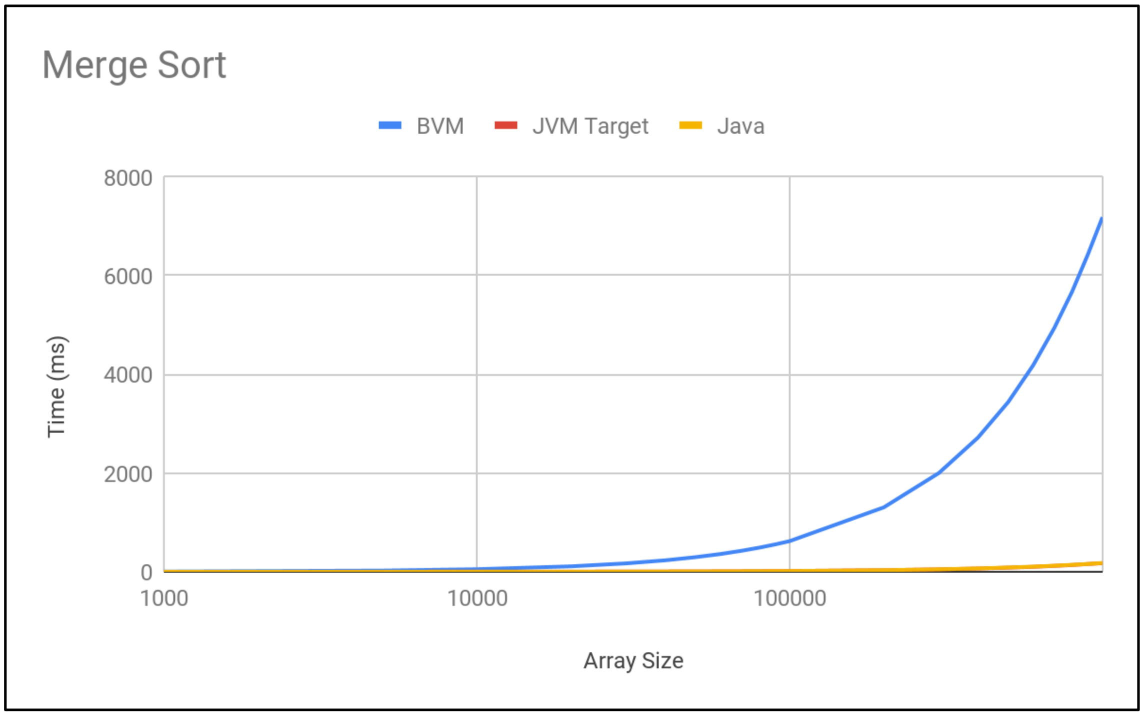


Figure 5.2 Mergesort Test

The observation here is that, BVM is growing exponentially lot sooner than JVM target. The JVM target is performing very close to the pure Java based implementation of the mergesort.

5.3 Quick Sort

The quicksort is written using the well known quick sort algorithm which uses a similar approach on using a recursive invocation to sort the given integer array. The array size was increased from 2000 to 10000 with the interval being constant value at 1000.

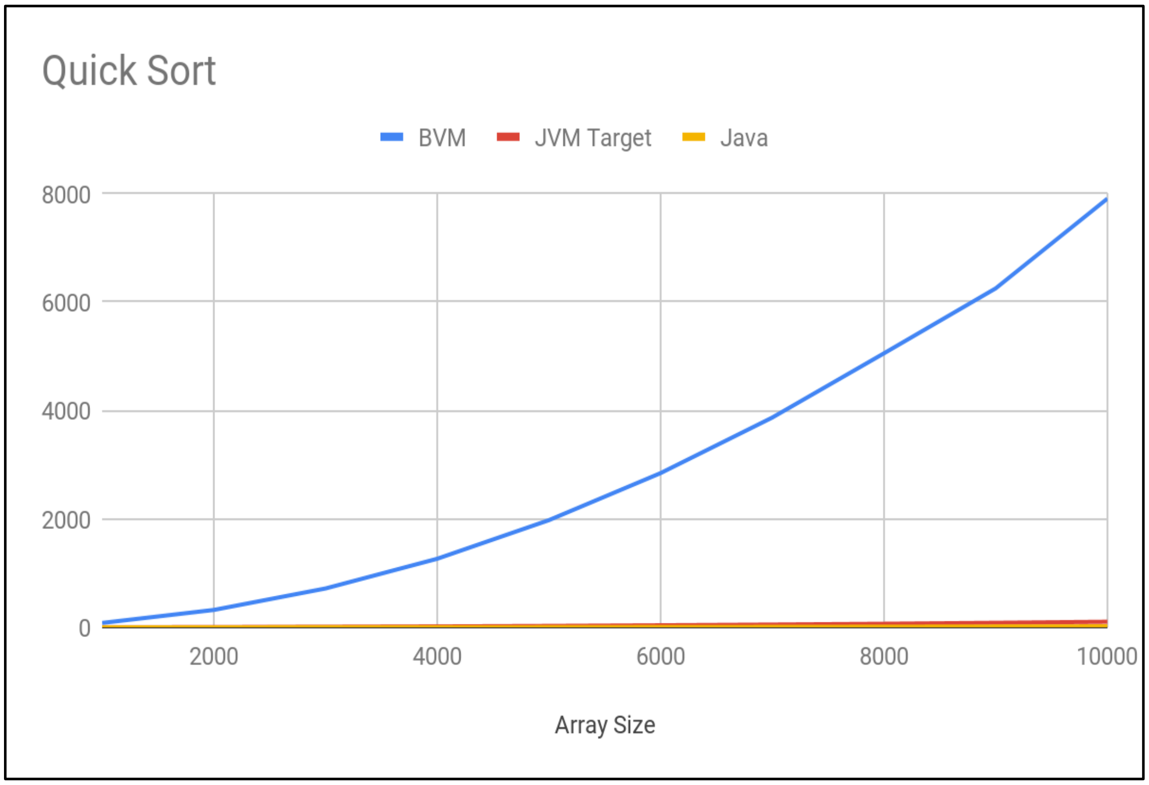


Figure 5.3 Quicksort Test

A similar observation is found in this test case also. With size of the array, the sorting time tends to grow exponentially for BVM. However the JVM compiler backend was performing very close to Java based implementation of the quicksort.

5.4 Matrix Multiplication

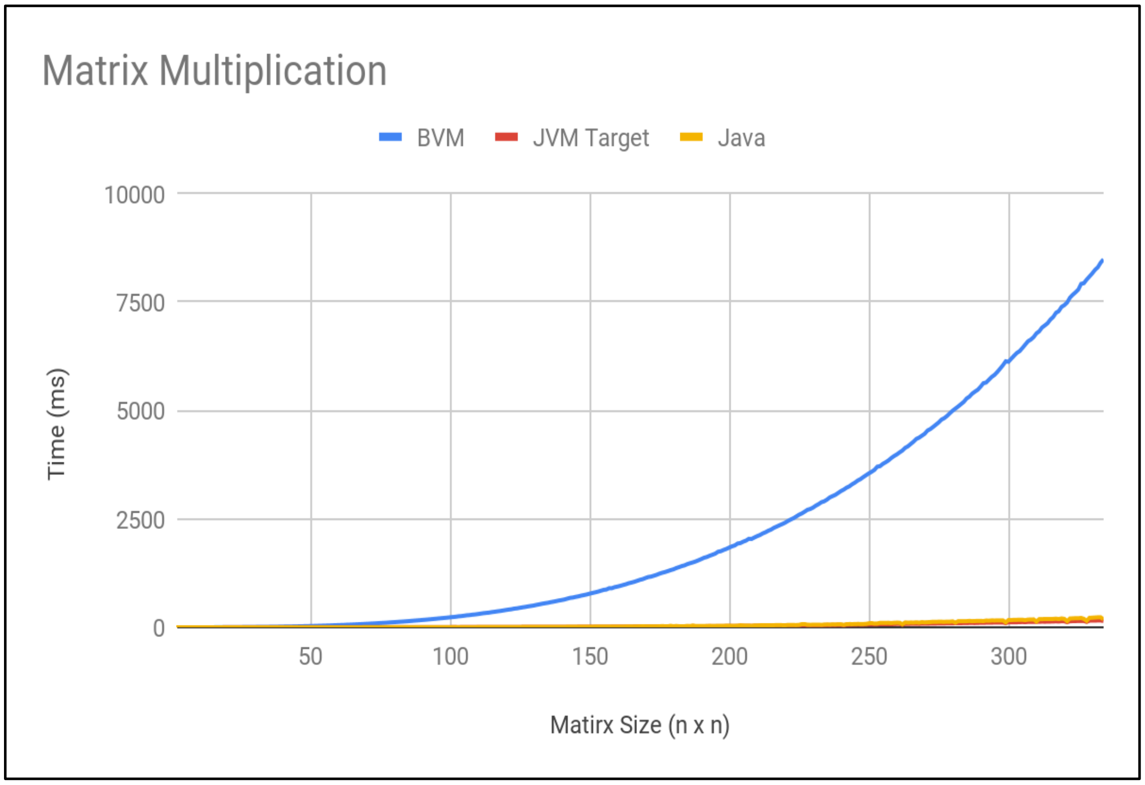


Figure 5.4 Matrix Multiplication Test

The matrix multiplication test uses two dimensional integer array multiplication logic where two arrays of same size was multiplied. The size of the two dimensional array (n x n) was started from 2 and the maximum array dimension was increased up to 350. The same dataset was used for all three types of runtimes (BVM, JVM Target and Java) and the time to complete the multiplication was measured.

5.5 String Regular Expression Match

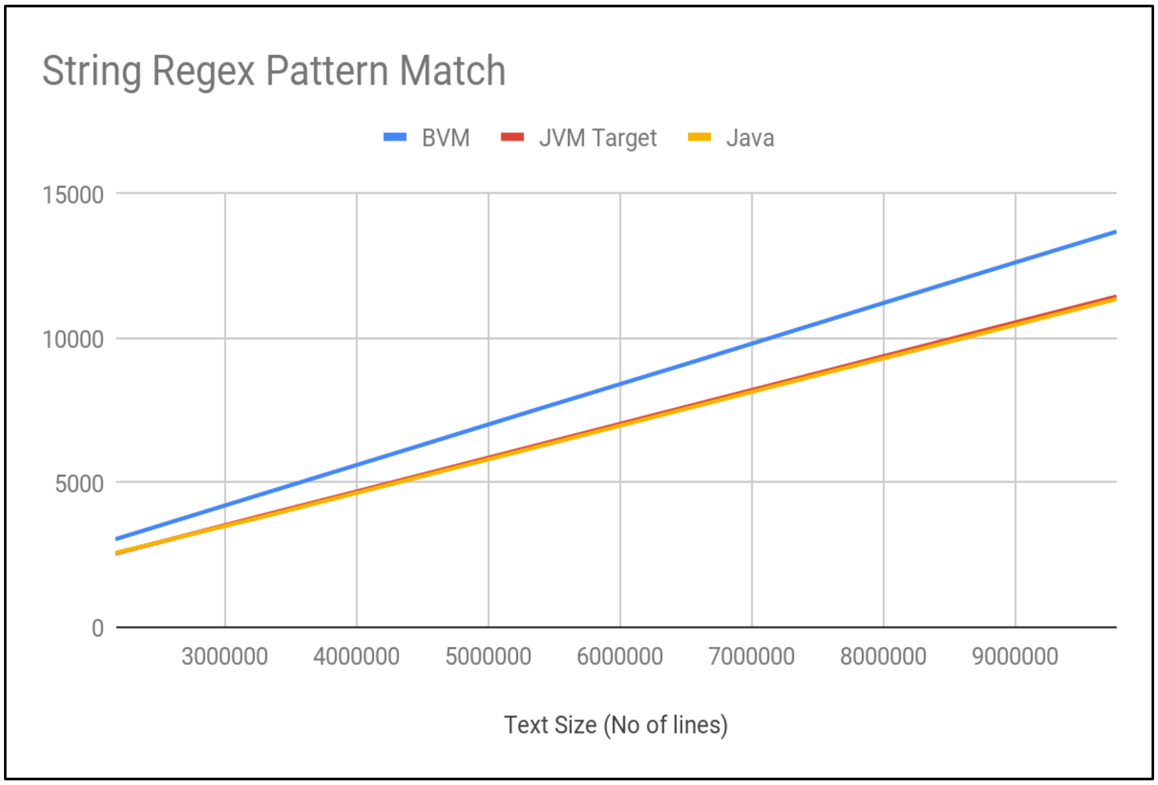


Figure 5.5 String Regular Expression Matching Test

This scenario uses a text file which has some text content such as paragraphs. A string regular expression pattern was used to match with the input text content. The text content of the sample text file was fed into the program and then the number of occurrences of the pattern was calculated and the time to complete was measured by increasing the text content size using a constant number.

There is a different observation for this scenario compared to others. The time is not growing exponentially, however it is increasing linearly for all three runtimes. The time gap between the runtimes also increasing linearly.

5.6 Evaluation of the results

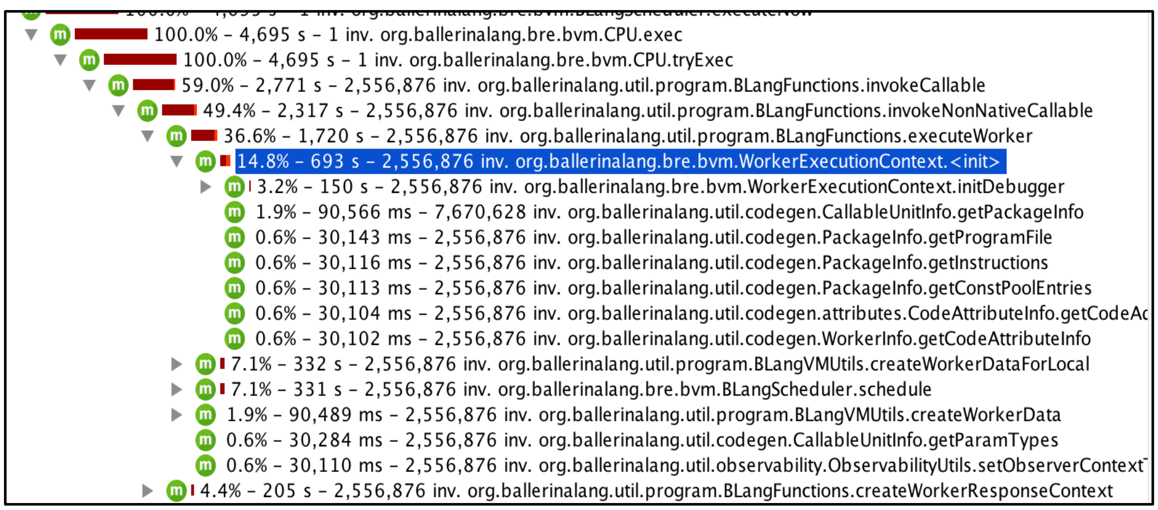
The main reason for ballerina runtime (BVM) performing low is that BVM is written on top of JVM as an interpreter. The ballerina bytecode is interpreted at runtime by the BVM and each of the bytecode instructions is executed one by one by the BVM. The core logic of the BVM is a while loop that does the execution of each ballerina bytecode instruction. Hence the BVM can be considered as a thick layer that runs on top of JVM which will incur low performance when compared to pure java bytecode execution. However there is a slightly different observation was made for string pattern matching related scenario where the time was not growing exponentially, however it was increasing linearly for all three runtimes. This is due to BVM underneath uses the same string pattern matching logic of Java and it calls the same methods from Java. The time gap between the BVM and JVM on this scenarios was mainly due to the thick layer of BVM runtime logic that interprets and the execute the ballerina bytecode instructions.

The other reason of BVM low performance is how the function execution is modeled by the BVM. Function invocations are the common instruction that can be found at most of the time ina ballerina program. Each function execution in BVM is started by the “call” bytecode instruction. For a function call, there is a ballerina level stack frame gets created, which is similar to the JVM stack frame that gets created of each method invocation. A ballerina stack frame contains the execution context information that will be used to store method arguments, local and intermediate variable values and results. The stack frame that gets created is scheduled for execution by the ballerina function execution scheduler. These runtime constructs, such as stack frame, execution contexts are basically java objects that gets created at runtime. Therefore there will be java object instance creation and memory consumption cost involved for each functions.

Apart from this another observation is that ballerina runtime (BVM) is performing badly when we have loops or recursion. Basically, when there is CPU intensive operations, BVM becomes slow as the interpretation and execution tends to slow down and adds up time for completion. For example, for a recursion scenario, The runtime takes lot of time to initialize execution context for each method invocation. This becomes a bottleneck and time grows exponentially when we have loops. However on the String based operations, both JVM and BVM performing similarly is that there are no loops involved and we are directly dealing with String objects and operations such as regex_match directly on the instances.

Also when we have loops or recursion, number of instruction to execute by the BVM also grows which will also have an impact on the execution time. Therefore at bottom line, the current BVM runtime is not an optimized implementation and also it does not optimize the compile code like how Java compiler does. I can include these finding also with the study.

Below is a profiled snapshot of BVM when it was running the fibonacci series. We can see that WorkerExecutionContext (which is the runtime context created for each method invocation) takes 37 % of the time, out of which 15% of the time is being spent on just initializing. This becomes bottleneck when we are running in loops or recursion.



```
▼ 100.0% – 4,695 s – 1 inv. org.ballerinalang.bre.bvm.CPU.exec
  ▼ 100.0% – 4,695 s – 1 inv. org.ballerinalang.bre.bvm.CPU.tryExec
    ▼ 59.0% – 2,771 s – 2,556,876 inv. org.ballerinalang.util.program.BLangFunctions.invokeCallable
      ▼ 49.4% – 2,317 s – 2,556,876 inv. org.ballerinalang.util.program.BLangFunctions.invokeNonNativeCallable
        ▼ 36.6% – 1,720 s – 2,556,876 inv. org.ballerinalang.util.program.BLangFunctions.executeWorker
          ▼ 14.8% – 693 s – 2,556,876 inv. org.ballerinalang.bre.bvm.WorkerExecutionContext.<init>
            ▶ 3.2% – 150 s – 2,556,876 inv. org.ballerinalang.bre.bvm.WorkerExecutionContext.initDebugger
              1.9% – 90,566 ms – 7,670,628 inv. org.ballerinalang.util.codegen.CallableUnitInfo.getPackageInfo
              0.6% – 30,143 ms – 2,556,876 inv. org.ballerinalang.util.codegen.PackageInfo.getProgramFile
              0.6% – 30,116 ms – 2,556,876 inv. org.ballerinalang.util.codegen.PackageInfo.getInstructions
              0.6% – 30,113 ms – 2,556,876 inv. org.ballerinalang.util.codegen.PackageInfo.getConstPoolEntries
              0.6% – 30,104 ms – 2,556,876 inv. org.ballerinalang.util.codegen.attributes.CodeAttributeInfo.getCodeAc
              0.6% – 30,102 ms – 2,556,876 inv. org.ballerinalang.util.codegen.WorkerInfo.getCodeAttributeInfo
          ▶ 7.1% – 332 s – 2,556,876 inv. org.ballerinalang.util.program.BLangVMUtils.createWorkerDataForLocal
          ▶ 7.1% – 331 s – 2,556,876 inv. org.ballerinalang.bre.bvm.BLangScheduler.schedule
          ▶ 1.9% – 90,489 ms – 2,556,876 inv. org.ballerinalang.util.program.BLangVMUtils.createWorkerData
            0.6% – 30,284 ms – 2,556,876 inv. org.ballerinalang.util.codegen.CallableUnitInfo.getParamTypes
            0.6% – 30,110 ms – 2,556,876 inv. org.ballerinalang.util.observability.ObservabilityUtils.setObserverContext
        ▶ 4.4% – 205 s – 2,556,876 inv. org.ballerinalang.util.program.BLangFunctions.createWorkerResponseContext
```

Figure 5.6 JProfiller based Profiled view of BVM

Therefore based on the above analysis, the current BVM based bytecode interpretation approach is not scalable when it comes to performance oriented and CPU bound scenarios. The BVM performance is low in those scenarios. This will become a bottleneck and showstopper for future of BVM based runtime approach for ballerina. Therefore the runtime has to be improved to have acceptable performance compared to the pure java based approach.

These performance limitation faced by the BVM can be overcome with the JVM compiler backend which generates the java bytecode directly for the ballerina source. When the java bytecode is generated and the ballerina source is compiled down as class file, the BVM will no longer be needed. The generated classes can now be directly invoked by the JVM. Also using JVM has some added advantages such as the bytecode gets optimized further by the JIT [16] optimization of the JVM.

6. CONCLUSION

This project presents an approach to compile ballerina source to Java bytecode directly. The initial performance results of JVM compiler backend seems promising and it can be considered as the possible alternative for the current BVM based ballerina runtime. The performance aspect will play a major role when choosing a technology, therefore improving the runtime performance of ballerina programs will become the key factor for the success and adaptation of ballerina. The ballerina program is aiming to solve integrations and microservices related problems that interacts with networked applications. Therefore, having a high performance runtime will make it a better choice among the programmers and developers who are looking to use ballerina language rich set of capabilities.

6.1 Limitations

The java bytecode generation is solely depend on the model that BIR produces. The jvm compiler backend does not directly depend on the ballerina source program and its semantics. Therefore, the BIR has to capture and model all the language level aspects and the ballerina runtime semantics which then can be used by any type of compiler backend implementation. By the time, that this project was implemented, the BIR model did not capture all the aspects of the ballerina language. So, the required constructs has to be added to the BIR model first and then generating bytecode for jvm was carried out. As the BIR model is still new and evolving, some of the constructs such as arrays and array operations has to be designed first with respect BIR constructs and added.

One of the other limitation with BIR model is that, it does not currently capture line number information and variable names. Therefore, when generating the jvm bytecode, the source file mapping and line number information mapping was not used. This support will be needed when adding the stack trace and debugging support for the jvm based compiler backend.

6.2 Future Work

As future expansion of this project, we can consider the list of things explained in here. Since this project focused on few language constructs of ballerina, as the next step, we can bring in other language constructs support with compiling to JVM target.

6.2.1 Reference types support

The reference types supported by ballerina type system such as records, maps, objects, XML, JSON, etc has to be modeled at JVM level. This modeling will probably involve generating new classes for these reference types along with some type descriptor level methods which should handle the type assignability and conversions related requirements.

6.2.2 Debugging support

The ballerina source file line number and variable naming information has to mapped to the generated JVM class files for debugging support. The java level debug can be used underneath to debug the ballerina source file since compiled java class files can be easily debug using java debugging support available at IDE's.

6.2.3 Error and stack trace modeling

The line number information mapping will also help with stack tracing in the event of error. Additionally, the error construct has to be mapped properly at JVM level as some exception class therefore operations such as panic and trap ballerina constructs are supported at JVM level.

6.2.4 Concurrency modeling

The worker model of ballerina concurrency construct has to be modeled at JVM level. This will be complete new area as the worker model is slightly different form a typical java threading model with work yielding and resuming support with the own scheduler implementation that could be needed at JVM level.

6.2.5 Update BIR model with all language constructs

The BIR model needs to capture all the ballerina language level semantics without any loss of information from ballerina source files.

Additionally there are other areas such as modeling the ballerina native functions invocation with JVM level, which can be considered as a similar approach that JVM used to invoke its native functions that are written in C++ language mostly. This will make ballerina to invoke java level code within ballerina programs that are known and understood by the compiler and the correct linking gets generated at runtime.

Optimizing the generated BIR instructions and also optimizing the generated java bytecode also can be considered for future work as this will optimize the instructions set with fewer instructions which would give added performance gain. This task will require code flow analysis to properly do an optimization.

7. REFERENCES

[1] J. Jones, “Abstract syntax tree implementation idioms,” in *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

[2] G. Hohpe and B. “Woolf. Enterprise integration patterns”. In *9th Conference on Pattern Language of Programs*, pages 1–9, 2002.

[3] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. “Bringing the web up to speed with WebAssembly”. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[4] N. Maurer and M. Wolfthal, “Netty in Action”. Manning Publications, 2016.

[5] Terence Parr and Russell Quong. “ANTLR: A predicated-LL(k) parser generator.” *Journal of Software Practice and Experience*, 25(7), 1995.

[6] C. Lattner, “Introduction to the LLVM Compiler Infrastructure,” in *Itanium Conference and Expo*, April 2006.

[7] J. Pombrio and S. Krishnamurthi. “Resugaring: Lifting evaluation sequences through syntactic sugar”. In *Programming Languages Design and Implementation*, 2014.

[8] C. Lattner. “LLVM and Clang: Next Generation Compiler Technology”. In *The BSD Conference*, May 2008.

[9] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis and transformation,” in *Proc. CGO*, 2004,

[10] Gosling, J. “Java Intermediate Bytecodes”. in *SIGPLAN Workshop on Intermediate Representations (IR95)*. 1995.

[11] D. Chisnall. “The Challenge of Cross-language Interoperability.” *Commun*. ACM, 2013.

[12] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. “Virtual machine showdown: Stack versus registers.” *ACM Transactions on Architecture and Code Optimization*, 2008.

[13] M. Wilde, M. Hategan, J.M. Wozniak, B. Clifford, D.S. Katz, I. Foster, “Swift: a language for distributed parallel scripting,” *Parallel Computing*, 2011.

[14] W. Binder, J. Hulaas, and P. Moret. “Advanced java bytecode instrumentation.” In *Proc. of the International Symposium on Principles and Practice of Programming in Java*, 2007

[15] B. Venners, “Inside the Java Virtual Machine, 2nd Edition”, McGraw Hill, 1999

[16] A.-R. Ald-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, “Fast, Effective Code Generation in a Just-in-Time Java Compiler,” Proceedings, *SIGPLAN ’98 Conference on Programming Language Design and Implementation*, 1998.

[17] “The Java® Virtual Machine Specification” [Online]. Available : https://docs.oracle.com/javase/specs/jvms/se8/html/index.html [Accessed: 16-March-2019].

[18] JVM Compiler Backend Benchmark [Online]. Available : https://github.com/Kishanthan/ballerina/tree/bir_jvm_benchmark/benchmarks/bir-jvm-benchmark/src/main/java/org/ballerinalang [Accessed: 16-March-2019].

[19] Ballerina Programming Language [Online]. Available : https://ballerina.io/ [Accessed: 16-March-2019]

[20] Ballerina Language Specification [Online]. Available : https://ballerina.io/res/Ballerina-Language-Specification-v0.990-2019-01-16.pdf [Accessed: 16-March-2019]

[21] Ballerina Compiler Architecture [Online]. Available : https://github.com/ballerina-platform/ballerina-lang/blob/master/docs/compiler/compiler-architecture.md [Accessed: 16-March-2019]

[22] Gary L. Schaps, “Compiler construction with antlr and java,” *Dr. Dobb’s Journal*, 1999

[23] Joe Groff and Chris Lattner. “Swift’s High-Level IR: A Case Study of Complementing LLVM IR
with Language-Specific Optimization.” *2015 LLVM Developers’ Meeting*, 2015.

[24] E. Bruneton. Asm 3.0, a java bytecode engineering library [Online]. Available : https://asm.ow2.io/asm4-guide.pdf [Accessed: 16-March-2019]

[25] JVM Compiler Backend Benchmark Test Results [Online]. Available : https://goo.gl/96W93J [Accessed: 16-March-2019]