AJVM – Java Virtual Machine Implemented in ActionScript 3.0

Submitted: 5th January 2016; accepted: 2nd February 2016

Arkadiusz Janik, Jakub Krawczyk

DOI: 10.14313/JAMRIS_1-2016/8

Abstract:
This paper describes the concept and implementation details of AJVM – state-of-the art Java Virtual Machine (JVM) implemented in ActionScript 3.0. Action Script is an objective programming language that supports compilation to Java bytecode. In the presented solution there has been a novel idea utilized – to use the other Virtual Machine's execution environment (ActionScript Virtual Machine) to build Java Virtual Machine. The subset of features specified in JVM Specification v.2 supported by AJVM has been chosen in a way which enables the machine to be used in many practical applications both in the commercial context as well as in science. As the architecture of AJVM is modular, the extension of its features in the future will not cause any difficulties. The implementation of AJVM in ActionScript 3.0 which is executed by ActionScript Virtual Machine (AVM) makes it possible to use Java code in applications written in ActionScript 3.0. It spawns many new opportunities considering that AVM is a part of FlashPlayer – commonly used multimedia player available in the form of plugins for the majority of modern web browsers, including mobile versions.

Keywords: *JVM, Action Script, bytecode*

1. Introduction

Traditional, commercial implementations of JVM Specification aim to increase the performance of a Virtual Machine (VM). As a result VMs are usually implemented in low level programming languages, tightly integrated with Operating System (OS) and hardware (Just In Time compilers that compile bytecode into native code of the hardware platform). The goal of AJVM is to execute Java applications in a variety of computers equipped mainly with a web browser, to increase code reusability by making it possible to use Java libraries in Flash applications and to provide convenient platform for end-users to observe and understand concepts behind JVM. On top of that the solution extends Flash platform with new features: multi-threading, blocking operations, generic classes as well as a new concept of executing bytecode.

1.1. Basic Terms

There are two major types of virtual machines: emulators and interpreters [9]. The emulator is a solution allowing to execute (in an isolated environment) the whole OS and other software designed for a specific platform due to hardware virtualization (VMWare Workstation or Oracle xVM VirtualBox). The interpreter is software capable of executing binary, precompiled code which is an output of a built-in compiler defining its own architecture as of a virtual device [7]. In the further part of this paper the term *virtual machine* will be used to denote the interpreter.

1.2. Flash Platform and ActionScript 3.0

Adobe Flash technology (previously known as Macromedia Flash) is a multimedia platform enabling new features in web pages: animations, video streaming, interactivity commonly used to implement games, advertisements as well as more sophisticated and complex applications.

Flash applications are published as SWF files (Shockwave Flash Object). Usually there is a single file per application even though it may contain many libraries and multimedia resources. In Flash 5 the concept of *actions* introduced in Flash 4 was extended and, for the first time, *Action Script* term was used in the context of a programming language [10, 11]. Flash 7 was released together with the second version of ActionScript programming language: such features as type-control (during compilation) and inheritance based on classes were introduced. Flash Players 9 was released with ActionScript 3.0 – the language was redesigned significantly and, to support downgrade compatibility, there are two virtual machines in the player: AVM1 (to support ActionScript 1.0 and 2.0) and AVM2 (to support ActionScript 3.0) [14]. The number of new features have been introduced to the player including performance optimization (hardware acceleration for DirectX and OpenGL), type-control (during code compilation and execution), separation of class-based and prototype-based inheritance, using packages, namespaces and regular expressions, new bytecode format, support for E4X format and others.

To sum up: ActionScript 3.0 is object-based, imperative programing language with strong type-control, compiled to bytecode being executed on AVM2 virtual machine, single-threaded (driven by events triggered by Flash Player) so not supporting blocking methods, with automatic memory management (Garbage Collector), not supporting generic classes, not supporting anonymous classes.

In the further part of the paper any references to *Action Script* will refer to Action Script 3.0.

2. Related Work

There are two major aspects of the contribution of our work:

Implementation of Java Virtual Machine in a non-standard environment

Emulation of Java platform inside Flash Player.

In the opinion of the authors, there are no solutions other than AJVM that handle both: Action Script and JVM. This makes AJVM a unique system capable of emulating Java platform inside Flash Player.

There are number of publications on the subject of non-standard JVMs including the ones mentioned below:

Jamiga – the goal of the project is to execute Java applications on Amiga computers [16].

JC – the property of the JVM is a novel approach to executing bytecode. All Java classes are translated on-the-fly to source code in C language and then compiled to native code thus enabling performance similar to traditional Just In Time Compilers (JITs) [17].

Squawk – implementation of JVM for Java ME (Micro Edition) for embedded systems and small, mobile devices. All elements, except for low-level modules supporting I/O operations and OS specific code, were implemented in Java (including Garbage Collector) [15].

GNU Classpath – Java Standard Library distributed under GNU license providing a great base to build own JVMs [20].

JOP – Java Optimized Processor is a hardware implementation of JVM with predictable execution time for embedded real-time systems. Due to the small size of the processor used, it can be implemented in a low cost FPGA (Field-Programmable Gate Array). For low volume systems, the flexibility of an FPGA can be of more importance though slightly more expensive than conventional processors. The processor was designed in VHDL programming language (Very high speed Hardware Description Language). The processor executes bytecode directly, without necessity to compile in-time nor to parse/interpret class files. Using FPGA allows JOP to dynamically declare stack size (which is consistent with JVM –stack-based VM rather than register-based) [21].

Sable VM – is a highly portable and efficient Java virtual machine, using state-of-the-art interpretation techniques. Its goals is to be reasonably small, fast, and compliant with the various specifications (JVM specification, JNI, invocation interface, etc.) [18].

2.1. Emulators of Other Platforms Implemented in the Flash

Even though ActionScript 3.0 is relatively new language there have been several emulators implemented so far:

FC64 – Flash Commodore 64 Emulator – FC64 is a low-level, fully functional emulator of Commodore 64 allowing a user to execute applications designed for Commodore 64 as well as to write code in BASIC programming language [22].

FlashZXSpectrum48k – Sinclair ZX Spectrum Emulator} – FlashZXSpectrum48k is a solution similar to FC64. The difference is that it emulates Sinclair ZX Spectrum platform.

AminNes – Flash NES Emulator – Flash NES Emulator is an emulator of Nintendo Entertainment System – 8 bits gaming console equipped with 2kB RAM and 2KB video memory. The emulator supports MOS 6502 and guarantees the highest quality of rendered video.

Flip8 – CHIP-8 Flash Emulator – Flip8 is an emulator of CHIP-8. CHIP-8 is a virtual machine designed in 70s used to interpret programming language called CHIP-8. It used to be installed on a graphical calculators. Flip8 is an emulator able to execute bytecode including 35 different operations [19].

3. Implementation of JVM in ActionScript 3.0

ActionScript Java Virtual Machine (AJVM) was implemented in Action Script [13]. It implements a subset of features described in the specification of JVM v.2.0. The sections included below present more information on the architecture and implementation problems the authors of this article had to solve.

3.1. Virtual Machine vs Standard Library

The crucial part of the implementation of any Java Virtual Machine is a standard library – the implementation of core Java classes described by JVM specification [4]. Even execution of the simplest *Hello Word* Java application requires hundreds of core Java classes to be present and loaded into VM. The solution described in the paper contains following elements in terms of a standard library:

AJVM library – ActionScript 3.0 library containing executable files of AJVM, ready to be used in any Flash or Flex projects. Flex is a set of components allowing a developer to write RIA applications. It is being compared to technologies such as XUL, JavaFX or Silverlight [12].

ART standard library – an equivalent of Java standard library included in any JVM distributions (such as JRE System Library for JavaSE). A standard library is specific for VM implementation as it expects native implementation of selected methods in the VM. For the purpose of AJVM we implemented a standard library called ART (ActionScript Runtime) which is based on GNU Classpath.

As the authors focused on the selected features of JVM, the ART is not a complete implementation of the standard library though it is possible to extend the implementation in future.

3.2. High Level Overview of AJVM

AJVM is available as an object created by the programmer's code inside his/her Flash application (in the similar way as for Jython interpreter for Python programming language). Consequently, the programmer's code has a full control over AJVM. Moreover, several isolated instances of AJVM within the single Flash host application can be created.

Due to the features of the Flash platform usually the code of Flash host-application and the code of AJVM are distributed in a single SWF file whereas Java classes that are part of ART and Java classes of the programmer's code will be distributed as JAR archives (or single *.class files) located on the same web server.

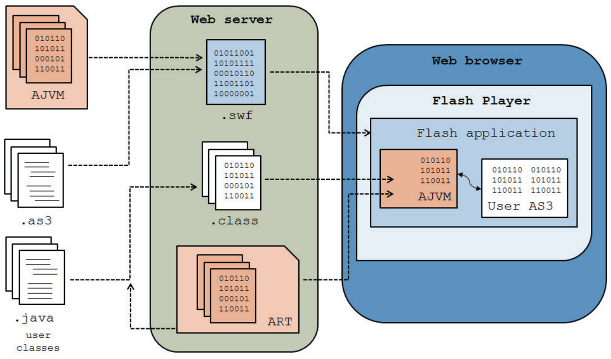


Fig. 1. High level overview of AJVM

Figure 1 presents a high level overview of AJVM.

As mentioned before, the programmer's Flash code has a full control over AJVM particularly in the below areas:

Starting and stopping AJVM instances – instances of AJVM are independent objects created, started and stopped in moments which are arbitrarily chosen during the lifetime of host Flash application.

Configuring classpath – similarly to traditional JVMs, also AJVM needs classpath [3] to be defined to know where to look for compiled Java classes. Unlike traditional JVMs, AJVM does not have access to the file system thus it needs to download class files from a web server. AJVM adds a prefix with the name of its own web server to compose URLs used to locate class code. For instance, if one assumes that AJVM is used in Flash application available under *http://domain.server.com/apps/test/index.html* and SWF file is available under *http://domain.server.com/apps/test/application.swf*. Flash host application defined the classpath as *java/rt;java/classes* and then executed Java class ***pl.edu.agh.test.Example***. It means that AJVM tries to load the class from one of the locations below (until the first successful try).

http://domain.server.com/apps/test/java/rt/pl/edu/agh/test/Example.class

http://domain.server.com/apps/test/java/classes/pl/edu/agh/test/Example.class

AJVM supports dynamic, on-the-fly addition of new entries to the class path.

Specifying main class – Each JVM starts the execution of the user's code following the method:

public static void main(String args[]);

defined in the class specified as JVM's argument. AJVM also expects the name of the main class to be provided (as one of the mandatory elements of the configuration of AJVM).

Configuring lifecycle of VM – As mentioned before Flash is a single-threaded environment. The programmer's code is executed solely in response to events triggered by Flash Player (a mouse click, a keyboard event etc.). It is expected that an event handler gives control back to the Flash Player as soon as possible to guarantee smooth execution.

As the consequence, AJVM has to work in very short time slots divided by pauses long enough to handle other no Java events and to render the next frame of Flash animation. A developer of a Flash application defines by themselves (usually statically, when compiling the application) frequency of refreshing the application's view in a web browser. Flash Player takes all available steps to support the requested FPS (Frames Per Second) parameter.

We implemented similar logic for AJVM – a programmer using AJVM defines (as one of the parameters of AJVM configuration) how many milliseconds per frame AJVM can execute before giving the control back to Flash application. AJVM monitors its execution time and breaks the execution of the bytecode immediately when the requested threshold is met. In future we plan to add adaptive mechanism inside AJVM so that it dynamically changes the parameter depending on complexity of Java and/or Flash code and available computational power (for instance to slow down execution of Java code during dynamic animations of a Flash application).

Implementing native methods – some methods delivered with Java standard library are native methods. Similarly, user's methods can also be native. AJVM uses native code to implement ART thus enabling users to declare and implement own native methods – written in ActionScript 3.0. As a result the user's applications can communicate with Flash applications.

Configuring logger – one of the goals of implementing AJVM was to provide platform that can be used by users to experiment with and learn about architecture of VMs. Monitoring is crucial thus we equipped AJVM in easy to use logger supporting four levels: SEVERE, WARNING, INFO and DEBUG.

3.3. Executing Bytecode in AJVM

One of innovations of AJVM is a fully objective approach to VM's major functionality which is executing bytecode. The overview on the standard location of bytecode execution module in a typical JVM's architecture can be found in the Figure 2.

In Java the bytecode is one of many attributes of a method. In JVM specification bytecode instructions are identified as numbers (0x00 till 0xca). An execution module of a traditional JVM's works on binary instructions kept in a method area memory and represented in an unchanged way comparing to the compiled bytecode. The only exception is replacing constant pool elements with symbolic references (and, of course JIT compilation to native code). There are

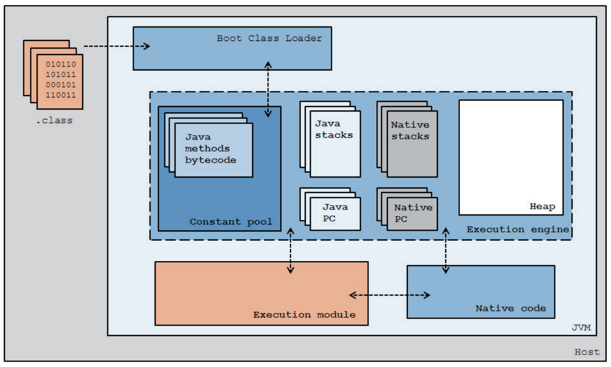


Fig. 2. Location of bytecode execution module in a standard JVM's architecture

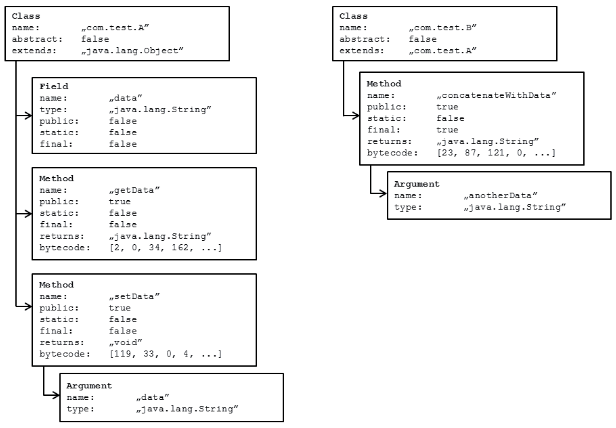


Fig. 3. Simplified version of syntactic parse tree for Java classes in AJVM

mobile implementations of JVMs utilizing XIP concept (eXecution In Place) thus not copying bytecode to the method area but referring to *.class files whenever the instruction is being executed [6]. There are following implications of the above mentioned models:

Necessity to interpret the instruction's code every time the instruction is executed – in Java's bytecode one needs to interpret the instruction to understand how many bytes after the instruction are the instruction's arguments, which is necessary to obtain the beginning of the next instruction.

Necessity to derefer pointers to constant pool every time the instruction is executed – the instruction's arguments can refer to classes, methods or fields addressed as entries in the constant pool (for instance: *new*, *anewarray* instructions which allocate objects or arrays, *invokevirtual* instruction which calls a method etc.).

The above approach may lead to a significant overload. For instance: getting a value of a non-static field of an object requires following steps to be executed:

Retrieve (from bytecode and using information from an entry in a constant pool) an address of a data structure with the description of the field,

Retrieve (from the field's description) an address of the data structure with the description of the field's class,

Retrieve an offset of the field using information about its class and its fully qualified name,

Retrieve from the data structure representing the object of the field the value of the field (using the offset of the fields and a data type of the field).

The above mentioned steps are not actioned for JIT compilation (JIT) [8].

The low performance of the standard approach to the bytecode execution and the access to constant pool data motivated us to look for another approach in which the following steps were taken:

The construction of objective representation of Java class – Construction of syntactic parse tree, which is an object representation of elements in class file (see Figure 3). The representation is not a typical graph representation of classes and relations between them. There are following known limitations:

Indirect references – Apart from direct relations between a class and its methods (or between a method and its arguments) a traditional syntactic parse tree stores relations between some objects purely by their identifiers. For instance: to retrieve any attributes of a class *com.test.B* one has to take string "*com.test.A*" and look for a graph of the class with such a name. Such an approach requires checking whether the object with such a name exists and this is time consuming.

Bytecode in a binary representation – A part of a syntactic parse tree – methods' bytecode – was not parsed at all. Inside instructions of a method there are references to classes, other methods and fields (which are also elements of syntactic parse tree), which remain in a binary representation as long as the bytecode is executed.

The above syntactic parse trees can be used to build even more effective representation shown in Figure 3. All relations between represented classes, methods, fields and even external types can be resolved immediately after the first step of class loading process (parsing the class representation) thus making execution of classes' bytecode more efficient.

Building object-oriented representation of the bytecode – As presented on the diagram from the previous section we decided to use object-oriented representation also for the bytecode. In a traditional approach the machine code is represented as a single-dimensional sequence of instructions executed one-by-one. In practice, instruction pointer very often changes in a more complex way due to branch instructions, conditional or unconditional jumps etc. The place in the bytecode that the control should be transferred to (after a branch instruction) is calculated as an offset relative to the current instruction pointer. It means that executing a branch instruction requires calculating the new address and reading the bytecode instruction from that address. The approach

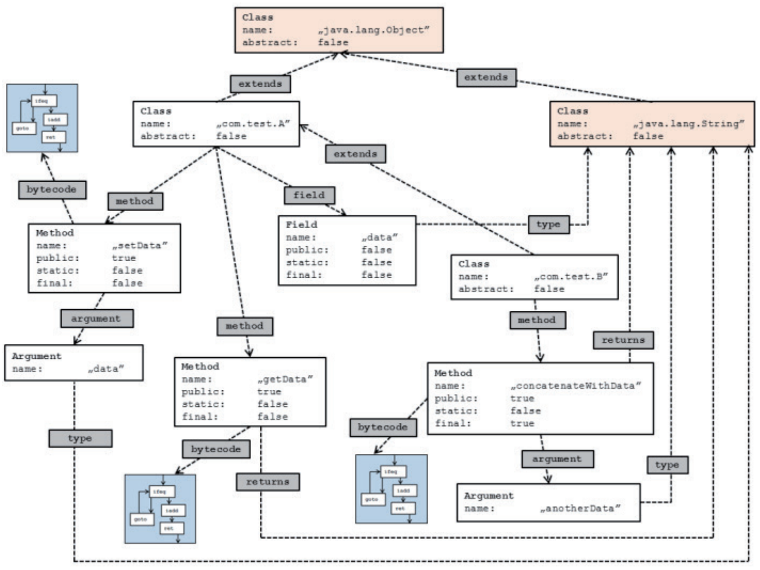


Fig. 4. A sample, full graph of classes and relations between them

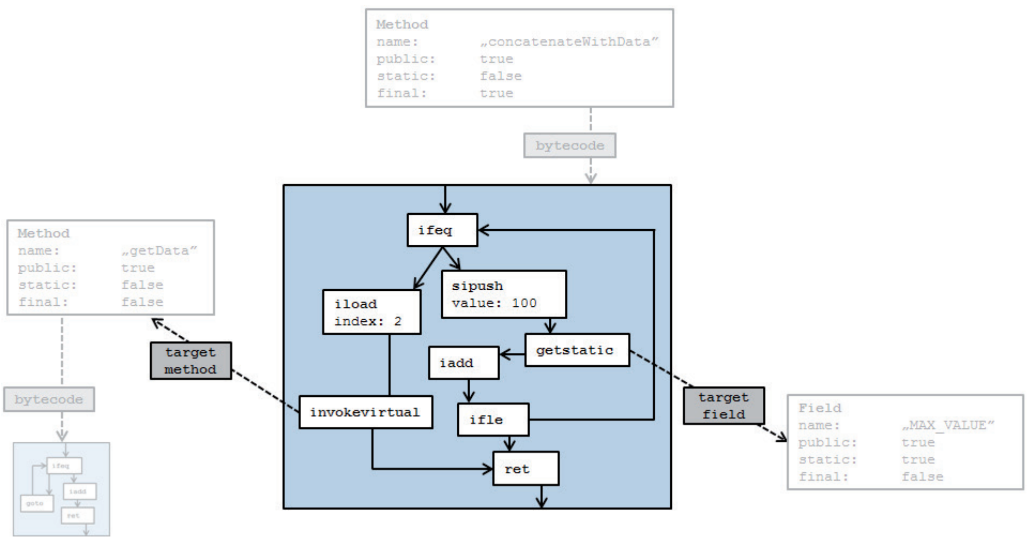


Fig. 5. Sample instruction execution graph shown in the context of objects it uses

we have taken is using a novel idea: after parsing the class, there is an instruction execution graph being built for each non-abstract method (see Figure 5).

Operation codes are mapped on classes of respective instructions. Whenever the instruction needs to be executed, the object's method is executed with the instruction's arguments. Each object has the knowledge (stored as reference to) about objects representing classes, fields and methods that are required by the instruction. Similarly, there are references to a consecutive instruction (or to consecutive instructions for branch instructions) stored in the object. The method stores the reference to the first instruction from its method whereas the bytecode stores the reference to its method so that it knows where to give control back when returning.

Delegation of executing an instruction to its object representation – Using an objective nature of ActionScript 3.0 language one can create full hierarchy of classes for Java instructions. A part of such a hierarchy is presented in Figure 6.

Each instruction implement method **execute()** declared in **Instruction** interface (design pattern **Command**). An argument of the method is object **FrameExecutionContext**, which represents a single stack frame. The object's diagram can be found in Figure 7.

FrameExecutionContext object provides access to computational stack and current values of local variables thus allowing an instruction to execute (itself) in the current context. After execution in the way specific for the instruction (for instance: pop two values from the stack, add them and push on to stack for **iadd** instruction) JVM has to set up the next instruction to be executed. As a result, conditional instructions can control their thread's execution. An instruction can use additional references passed when the class was parsed and the graph was built. For instance: class **JgetfieldInstruction**, which is representing **getfield** instruction stores the reference to an object of **JVMField** class, allowing it to load the current value of a field.

This is a decentralized approach: each of more than 200 bytecode's instructions has their implementation (in the **execute()** method) thus the conceptual schema of AJVM (see Figure 8) differs significantly from the traditional implementations of JVM.

The bytecode execution module in AJVM is virtualized. The module is created by instructions of loaded bytecode expressed as objects. It means that AJVM neither contains nor uses binary representation of bytecode. There are following advantages of such an approach:

One-time interpretation of bytecode – Binary representation of bytecode is read by AJVM once, when the class is loaded and the instruction graph is built.

Novel implementation of reflection – All classes, methods, fields and their attributes have objective representations thus the implementation of reflection

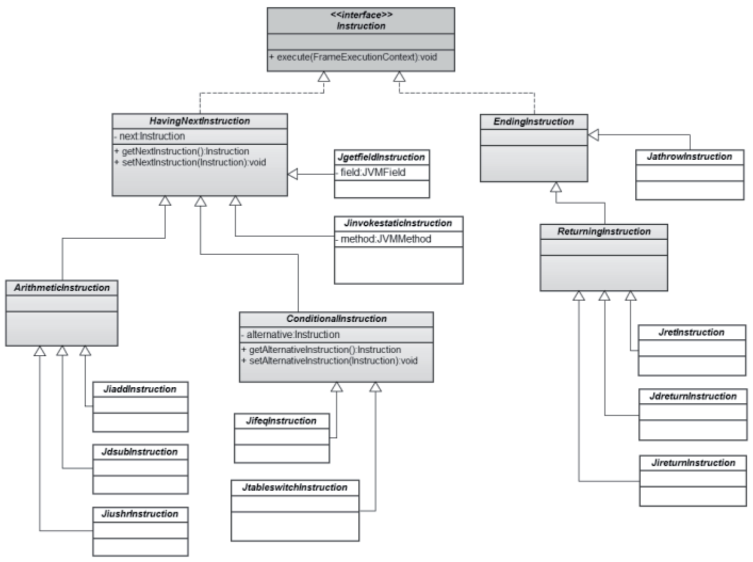


Fig. 6. A part of a diagram showing objective hierarchy of Java instructions

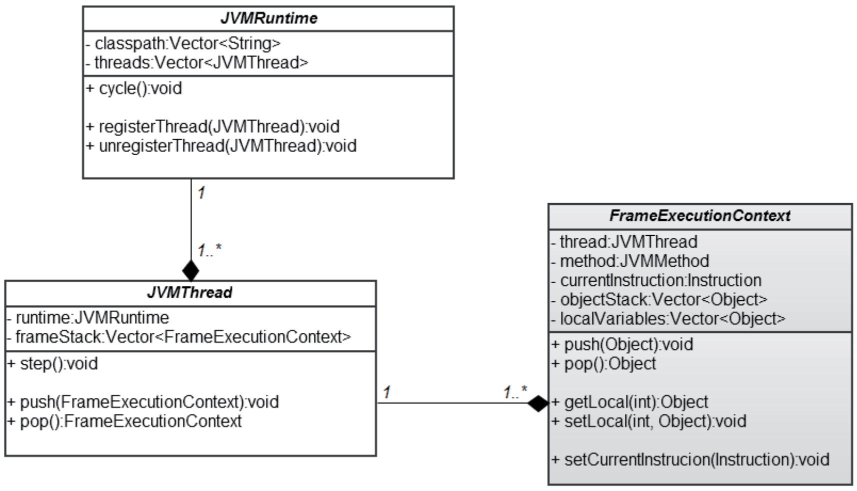

JVMRuntime
- classpath:Vector<String>
- threads:Vector<JVMThread>
+ cycle():void
+ registerThread(JVMThread):void
+ unregisterThread(JVMThread):void
1
1..*
JVMThread
- runtime:JVMRuntime
- frameStack:Vector<FrameExecutionContext>
+ step():void
+ push(FrameExecutionContext):void
+ pop():FrameExecutionContext
1
1..*
FrameExecutionContext
- thread:JVMThread
- method:JVMMethod
- currentInstruction:Instruction
- objectStack:Vector<Object>
- localVariables:Vector<Object>
+ push(Object):void
+ pop():Object
+ getLocal(int):Object
+ setLocal(int, Object):void
+ setCurrentInstrucion(Instruction):void

Fig. 7. A part of a diagram of the most important classes of AJVM execution module

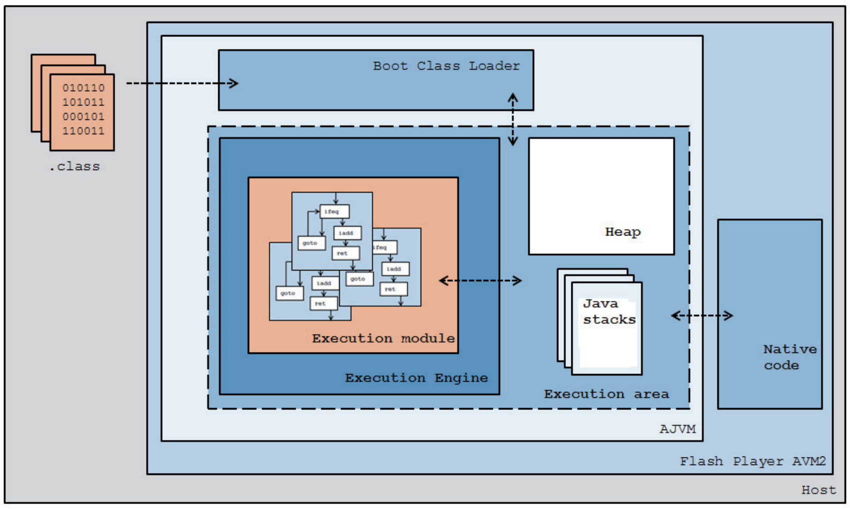


Fig. 8. Bytecode execution module in AJVM

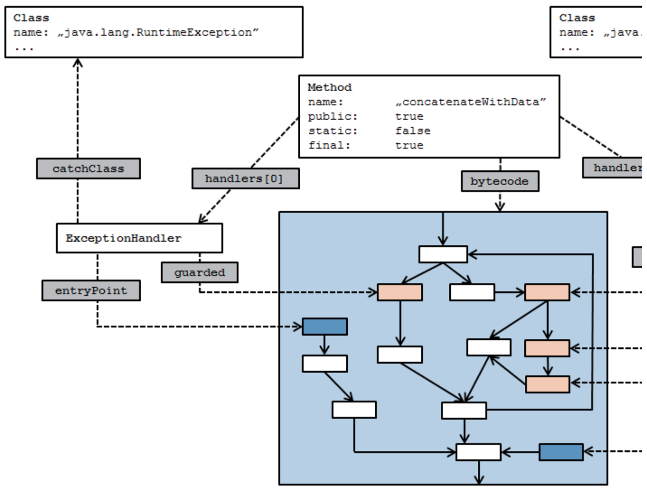


Fig. 9. An objective representation of an exception handler table for a method

does not require referencing native code.

Clear and flexible architecture – Inspection of the instruction graph as well as dependencies between classes make possible to extend AJVM in the future with interesting features such as dynamic code transformation or visualization of code execution which can be used to analyze and optimize code performance.

4. Additional Remarks on AJVM Implementation

4.1. Implementing Exception Handlers

One of the attributes of a compiled class is an exception handlers table. The table is used to store information about the class of an exception being handled, the first instruction being protected by the handler, the offset of the last instruction being protected and the offset of the first instruction of the exception handling procedure.

In AJVM there are no instruction offsets at all, as they are replaced by the instruction graph. Consequently exception handlers have to be defined in an objective way. Each method has a collection of objects representing exception handlers (the list may be empty for methods without try-catch blocks). An object representing an exception handler stores references to all instructions being protected (as it is impossible to represent the set as the first instruction and the offset). The idea is presented in Figure 9.

4.2. Garbage Collection

We solved the problem of Automatic Memory Management (AAM [5]) using features of ActionScript 3.0 which is equipped with AMM. Flash VM (AVM2) is equipped with Garbage Collector thus the easiest way of solving GC related problems is to:

- **Resign from our own heap in AJVM** – our JVM does not use heap at all.
- **Encapsulate each and every Java object inside native ActionScript objects** – it is possible due to representing each Java object as an instance of class **JVMObject** defined in ActionScript 3.0 allocated on Action Script's AVM2 heap and managed by AVM2. Each

Java object is encapsulated in ActionScript 3.0 object. Each Action Script's **JVMObject** stores a reference to the object representing the class of the Java object and a data structure to store the Java object's state (value of all fields from the object's class).
- **Utilizing AVM2 Garbage Collector** – as Java objects inside AJVM are represented by ActionScript's object they are managed by AVM2's Garbage Collector. The only limitation is not to store any additional, external references to these objects even though it may be useful for the purpose of VM monitoring etc.

4.3. Multithreading

The traditional implementation of JVMs provides multithreading through the mapping of Java threads onto native threads (on the Operating System level) [1, 2]. AJVM does not have access to low level OS threads as it is limited by Flash Player it is run in and which executes ActionScript 3.0's code just inside the event-handler's thread. Consequently, it was necessary to implement the multithreading model of our own design at the level of application. We had to use an objective representation of a thread as a part of VM. An instance of AJVM holds the collection of instances of **JVMThread** class. New objects are added to the collection every time a bytecode instruction executes **java.lang.Thread.start()** method, and are removed while returning from **java.lang.Thread.run()** method. As soon as the collection is empty the AJVM stops.

AJVM works in cycles and each cycle is about executing an amount of bytecode's instructions of each active thread. The length of AJVM's cycle (measured as a period of time as opposed to a number of instructions) is AJVM configuration item. The available time of each cycle is split between all threads accordingly to their priorities. The simplest way to meet the above requirements is to implement RoundRobin algorithm with the handling of priorities. Each AJVM cycle consists of/requires the following steps:

Calculate the possible time of the end of the cycle **endTime** (current time + configured cycle time).

If **endTime** is reached, give the control back to event handling procedure (thus finishing the cycle).

For each thread in the collection:

Execute method **step()** a number of times basing calculations on the priority of threads.

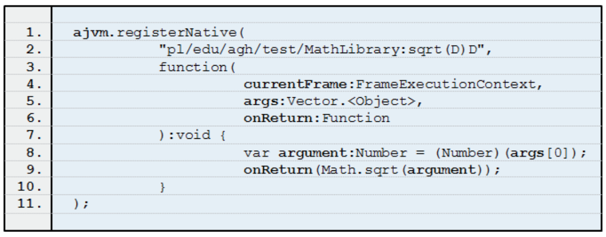
If the thread ends(execution stack is empty), remove the thread from the collection.

Go back to step 2.

Obviously, the above algorithm does not apply to "blocked" or "waiting for monitor" (**monitorenter** instruction) threads.

4.4. Native Interface

Native interface of AJVM allows a user to provide their implementation of Java classes as functions of ActionScript 3.0. It is done during VM initialization thanks to the method which binds a native method (its class, name and signature) with ActionScript 3.0 object that represents the closure of the function (Callback design pattern). Figure 10 demonstrates sample implementation and the registration of native method **sqrt**) in VM's case.



```
ajvm.registerNative(
        "pl/edu/agh/test/MathLibrary:sqrt(D)D",
        function(
                currentFrame:FrameExecutionContext,
                args:Vector.<Object>,
                onReturn:Function
        ):void {
                var argument:Number = (Number)(args[0]);
                onReturn(Math.sqrt(argument));
        }
);
```

Fig. 10. A sample code used to register a native method in AJVM

Arguments of the function are used to pass following elements from native interface of AJVM to the user's method:
- The context of the current execution frame in the execution stack frame – the reference to the case of **FrameExecutionContext** class that gives the access to properties of the current Java thread as well as to the stack and the local area of the stack frame. It allows the native method to use information that is forbidden for Java code (for instance to implement reflection with the use of the objective representation of classes and methods).
- The values of arguments passed to Java method. **args** contains 0 or more arguments passed to the native method by the Java code. Primitive types are mapped onto corresponding types of ActionScript 3.0 whereas objects are represented by their encapsulations (**JVMObject**) – in the native code one can easily get their classes and access fields (also private) in a way similar to the one used in the reflection.
- Callback that returns from native method – even though keyword **return** is present in ActionScript 3.0, employing it for returning from function would significantly reduce the capabilities of native methods. Each native implementation would have to give control back to AJVM soon after getting it as Flash Player uses a single thread. Implementing return functionality in the described way allows a developer to implement blocking methods (for instance input/output operations). A thread calling a native method is blocked until the callback is executed. In practice it means that even void methods (not returning anything) should contain **onReturn(null)** call at the end.

5. Test Cases

The output of the implementation part of our work includes an ActionScript 3.0 library implementing the most important features of Java Virtual Machine. The quality of the Virtual Machine can be measured with use of many indicators including the two most important: compatibility with JVM specification version 2 and performance. As was mentioned at the beginning of this paper, the full compliance with JVM specification was not our objective. Instead, we focused on those parts of Standard Java Library that allow a developer to create the majority of typical Java applications.

Neither high performance was the goal of AJVM. However, it is worth comparing the implementation of AJVM with the most popular JVM – Oracle HotSpot.

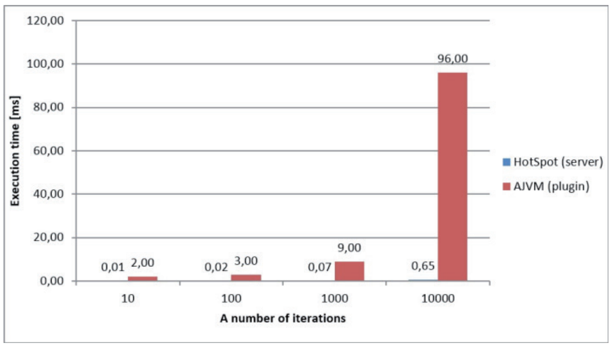


Fig. 11. Execution time of the test 1 – performing fixed point arithmetic

The following testing platform was used: (VM: Java HotSpot(TM) 64-Bit Server VM (build 21.0-b17, mixed mode, library: Java(TM) SE Runtime Environment (build 1.7.0-b147). AJVM (plugin) – VM: AJVM, reference version, library: ART, reference version, Flash Player: 11.2.202.233, ActiveX plugin, Internet Explorer 8. All tests have been executed on Intel(R) Core(TM) i7-2630QM @ 2.0 GHz, 8 GB RAM, System Windows 7 64-bit.

5.1. Test Case 1 – Fixed Point Arithmetic

The test was to iterate through an array of integers and perform several calculations on each element of the array. The results are presented in Figure 11.

Test results leave no doubts with regards to the differences between commercial HotSpot VM and experimental, research AJVM. Due to Just In Time compilation used in HotSpot the code is compiled to native representation and executed on a physical processor rather than on a virtual machine. As a result, calculations are done significantly faster.

5.2. Test Case 2 – Sorting of Floating Point Numbers

The test was to call **java.util.Arrays.sort()** method of the standard library to sort an array of double numbers. The results can be found in Figure 12.

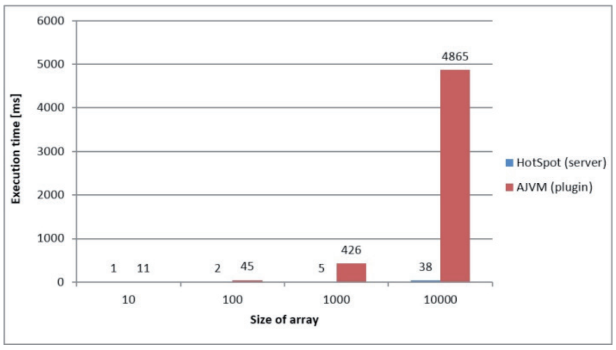


Fig. 12. Execution time of the test 2 – sorting arrays of fixed point numbers

Again HotSpot VM beats AJVM and again the major factor is JIT implemented in AJVM. Another important factor is that source code for arithmetic in HotSpot is written in C which makes development optimization techniques possible. It is worth mentioning that execution time increases linearly in a function of array size for both: AJVM and HotSpot virtual machines.

5.3. Test Case 3 – Sorting of Linked List of java.lang. Comparable Objects

The test was to call **java.util.Collections.sort()** method from the standard library to sort linked list (**java.util.LinkedList**) containing objects implementing **compareTo()** method. The test validates pointer operations and virtual methods (unlike sorting arrays of primitive types). The results can be found in Figure 13.

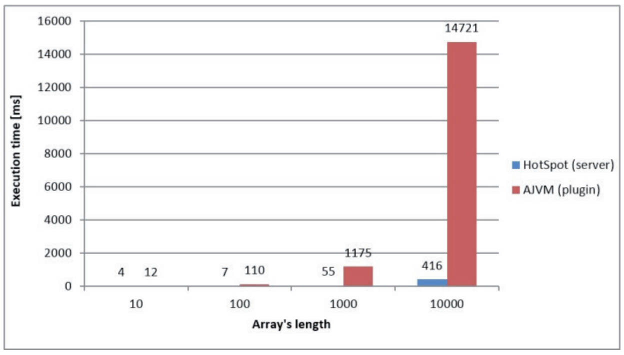


Fig. 13. Execution time of the test 3 – sorting linked lists of comparable objects

One can see an interesting observation here: even though both tests; test 2 and test 3 are about sorting of a collection, the increase of execution time in test 3 is significantly greater than in test 2. The reason is that in test 3 objects are sorted rather than numbers which results in more object operations (*compare()* method calls). And pointer operations are less efficient in AJVM than in HotSpot.

5.4. Test case 4 – Network Connections and Object Deserialization

The aim of the test was to establish a network connection with a remote server with the use of **java.net. Socket** so as to obtain some data and deserialize it. The results can be found in Figure 14.

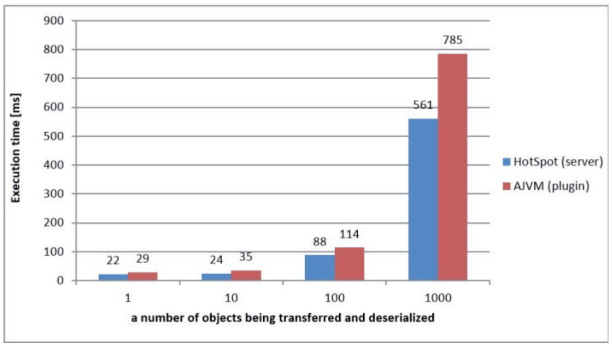


Fig. 14. Execution time of the test 4 – transferring data through network connection and deserializing it

Interestingly, the difference between execution time between AJVM and HotSpot is less visible and do not exceeds 35%. The reason is that in the test external, system resources are used (network connection) intensely. In other words: it is not so important how elements of a virtual machine are implemented as most operations are done on a level of OS. In other words, a proportion of input/output operations to other operations is higher than in previous tests.

5.5. Test Case 5 – Massive Multithreading

The test consisted of launching the number of concurrent threads, executing them and waiting for the last one to end. The results are illustrated in Figure 15.

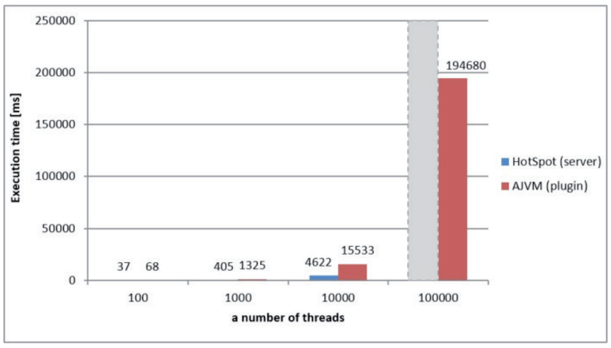


Fig. 15. Execution time of the test 5 – executing significant amount of concurrent threads

The grey bar in the Figure 15 for HotSpot indicates missing results – it was impossible to start so many threads.

The major factor that contributes to smaller execution time in AJVM comparing to HotSpot is a different threading model used in both solutions. HotSpot, like many other VMs maps Java threads to native OS threads. As a result thread creation is time-consuming operation. When there is a huge amount of threads OS may behave unstable. AJVM emulates multithreading programmatically – creating and starting a new thread is as expensive as creating one more Java object.

5.6. Test Cases Summary

The conclusions drawn from the executed test cases are as follows:

Tests 1, 2 and 3 leave no doubts with regards to the differences between commercial HotSpot VM and experimental, research AJVM. Due to Just In Time compilation used in HotSpot the code is compiled to native representation and executed on a physical processor rather than on a virtual machine. As a result, calculations are done significantly faster.

Test 4 shows more balanced results due to the significant influence of input/output operations in comparison with the execution time. Input/output operations are less CPU demanding thus AJVM is more effective. The results prove clearly that in some applications AJVM can be used with success.

Test 5 provides solid evidence in favour of the existence of domains on which AJVM is more robust than traditional VMs. The benefit of AJVM is the way thread creation is implemented. Having said that it must be emphasized that efficient creation of a thread does not mean that all thread operations are more effective on AJVM. As it was mentioned before test 5 focuses only on thread creation rather than on measuring all thread-related operations (creation, sleeping, waking up, thread scheduling etc.).

5.7. Samples of AJVM Utilization

This section demonstrates sample applications of AJVM. It describes how AJVM can be used to execute sample Internet applications – multiuser, text chat room.

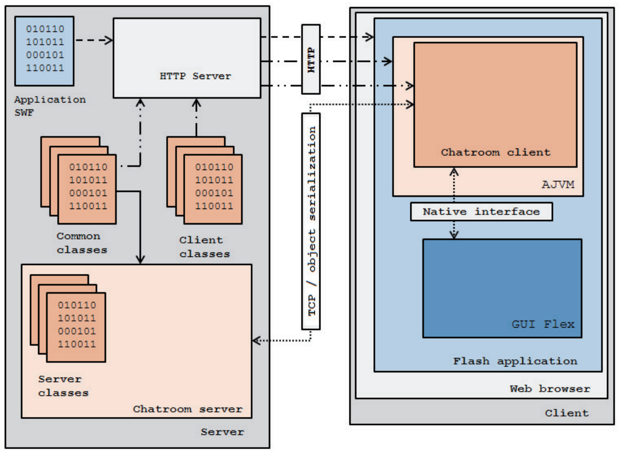


Fig. 16. The architecture of chatroom application executed on AJVM

The architecture shown in Figure 16 contains the following elements:

- HTTP server – any web server to provide static files via http protocol.
- Chatroom server – server implemented in Java for real-time, multi user chat room (Java Object Serialization is used).
- Chatroom client – client application written in Flash using Flex components. AJVM is applied to implement networking functionality (object (de) serialization and transfer) as well as implement business logic (logging in, authorizing and validating data). It is worth mentioning that the above sample uses Java code shared between the client and server applications: the same classes are loaded by HotSpot JVM (server side) and AJVM (client side). Once SWF file (Flash application) and Java classes are loaded, HTTP protocol is no longer used and there is a single network communication (TCP) established.

6. Conclusions

Authors of this paper successfully implemented Java Virtual Machine in ActionScript 3.0. As a result it was possible to use elements of existing elements of a virtual machine's infrastructure (FlashPlayer) to build your own solution. It allowed authors to focus on selected modules of a virtual machine (like implementation of a novel concept of bytecode execution engine) and to use existing ones which are beyond the scope of interest (GC, threading model etc.). Selection of an environment that supports Java bytecode (Action Script 3.0 and Flash Player) spawns many opportunities considering the fact that Flash Player is commonly used in most of modern web browsers.

As shown in the previous section, AJVM can be successfully used in implementing various applications, including Internet ones. There are many advantages of using AJVM including: **High reusability of classes**

Client and server elements of an Internet application usually use the same data structures, similar validation rules (for input data) as well as some common elements of business logic. In a traditional solution many of these common features are implemented twice (often using different implementation platforms): on each side. It is a developer's responsibility to guarantee consistency of both parts of the application. The use of AJVM enables developers to use the same Java code at the client's and server's side. It is a huge time-saver and the way to reduce a number of errors.

Providing lightweight implementation of Java Virtual Machine together with an application

Even though using Java applets meet the previously mentioned criteria (high reusability of classes), it is worth mentioning that Flash Player is much lighter and more popular than Java plugins for web browsers. AJVM makes it possible to execute Java code by users who do not intend to install Java Runtime Environment neither additional plugins to web browsers. End users do not have to be even aware of JVM running on their computers. AJVM is extremely lightweight – the size of compiled AJVM is less than 100kb.

Eliminating limitations of traditional clients

Client applications are usually running in single-threaded environment and thus use only a single thread (designed to support user's interface). Good examples are JavaScript, HTML5 and Flash. AJVM allows to use (on a client side) additional computational models: multi-threading, generics etc.

Of course, the performance of reference implementation of AJVM requires ActionScript 3.0 code to be responsible for such tasks as: UI rendering, multimedia streaming, and CPU intensive computations.

Executed test cases shows clearly that the current implementation is missing optimization techniques which speed up bytecode execution, particularly Just In Time compilation. Consequently, the efficiency of JVM is lower comparing to traditional, commercial VMs (represented by HotSpot in test cases executed). The difference of efficiency ranges from ~35% for test cases using intensively external resources (less operations on VM-level comparing to a number of operations on OS-level) to ~12 800% for others. The exception is some specific situations (described in test case 5) which reveals dominance of AJVM.

The future work on AJVM includes the extension of the compliance with the reference specification of JVM (for instance providing bytecode verification during class loading).

One should also remember that in order to seriously think about practical application of AJVM it should be made compliant with Java 7 or at least Java 6 standard.

Another significant work to be done is further development of Java Standard Library delivered with JVM (to provide full compliance with Oracle implementation or full compliance with GNU Classpath).

Another part of the research will concern increasing the performance. There are many options including Just-In-Time compilation. Also, the novel bytecode execution model opens new possibilities related to on-the-fly code transformation, code injection during runtime, and more efficient and precise code profiling.

As it was mentioned a couple of times, AJVM is designed in a modular way so that the level of dependency on external VM (Flash Player) can be reduced to some extent. For instance, it is possible to implement own Garbage Collector module and replace the one provided by Flash container. However, due to module dependencies, replacing a module may result in being forced to replace others as well. GC module mentioned above is a good example. In the current implementation objects creation and management is controlled by Flash at all. In order to replace GC both domains would have to be delivered.

Source code of AJVM can be accessed here:
http://galaxy.uci.agh.edu.pl/~ajanik/AJVM_source_files.zip

AUTHORS

Arkadiusz Janik* – AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. Mickiewicza 30, 30-059 Kraków, Poland. E-mail: arkadiusz.janik@agh.edu.pl

Jakub Krawczyk – AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. Mickiewicza 30, 30-059 Kraków, Poland.

*Corresponding author

REFERENCES

[1] Lindholm T., Yellin, F., *The Java Virtual Machine Specification*, 2nd Edition, Addison-Wesley, 1999.
[2] Gosling J., Joy B., *The Java Language Specification*, Addison-Wesley, 1996.
[3] Chan P., Lee R., *The Java Class Libraries: An Annotated Reference*, Addison-Wesley, 1997.
[4] Naughton P., Morrison M., *The Java Handbook*, Osborne/McGraw-Hill, 1996.
[5] Venners B., *Inside the Java 2 Virtual Machine*, McGraw-Hil, 2000.
[6] Downing T., Meyer J., *The Java Virtual Machine*, O'Reilly Media, 1997.
[7] Craig I., *Virtual Machines*, Springer, 2005.
[8] Stark R., *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer, 2001.
[9] Barrio V., Fernandez A., *Study of the techniques for emulation programming*, Universidad Politecnica de Catalunya, 2001.
[10] Braunstein, R., *ActionScript 3.0 Bible*", Wiley, 2010.
[11] *ActionScript Virtual Machine 2 (AVM2) Overview*, Adobe Systems Incorporated, 2007.
[12] Gassner D., *Flash Builder 4 and Flex 4 Bible*, Wiley, 2010.

[13] Elst P., *Object-Oriented ActionScript 3.0*, friendsofED, 2007.
[14] *Adobe Flash Player Technology Breakdown*. http://www.adobe.com/products/player_census/flashplayer/tech_breakdown.html accessed on 1st Oct 2015.
[15] Simon D., Cifuentes C., *The squawk virtual machine: Java on the bare metal*, ACM, 2005.
[16] JAmiga VM homepage, 2014. http://jamiga2.blogspot.com/ accessed on 15th July 2015.
[17] JC Virtual Machine homepage, 2013. http://jcvm.sourceforge.net/ accessed on 15th July 2015.
[18] Pickett C., Verbrugge C., *Return Value Prediction in a Java Virtual Machine*, VPW2, 2004.
[19] Flip8 – CHIP-8 Flash Emulator homepage, 2014. http://sourceforge.net/projects/flip8/accessed on 21st Sep 2015.
[20] GNU Classpath homepage, 2014. http://jcvm.sourceforge.net/ accessed on 21st Sep 2015.
[21] Schoeber, M., "JOP: A Java Optimized Processor for Embedded Real-Time Systems", VDM Verlag Dr. Müller, 2008.
[22] FC64 – Flash Commodore 64 Emulator homepage, 2014. http://codeazur.com.br/stuff/fc64_final/accessed on 2nd Apr 2015.