Volume 3, Issue 5, May 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Comparative Study of JVM

Amar Paul Singh
School of CSE
Bahra University, Shimla Hills, India

Rajneesh Gautam
School of CSE
Bahra University, Shimla Hills, India

Dhanshri Parihar
School of CSE
Bahra University, Shimla Hills, India

Abstract- This paper describes our work in progress on the working and architecture of Java Virtual Machine (JVM) infrastructure. Our research on JVM architectures that deliver scalable high performance for scientific applications on large numbers of processors. We describe the basic architecture of JVM; JVM provides "Compile once and Execute anywhere" technique. This research paper also includes comparison of 4 different virtual machines.

Keywords- Oak, byte code, optop, CORBA, LDAP, WORA.

I. INTRODUCTION

A. Objective: -

The objective of this research work is to describe the architecture of JVM ,and also includes comparison of JVM with other Virtual machine .

B. *Scope of JVM: -*

JVM provides high performance for distributed applications and optimized for various architectures, by ensuring reliability, manageability and flexibility for java application. There is a scope in JRE (Java Runtime Environment).It provides a cross platform functionality to Java.

C. *Java:-*

Java is an object oriented programming language developed by James Goslings, Patrick Naughton, Chris warth, Ed fronk and Mike Sherfidan in Sun Microsystems inc. in 1991. It took 18 months to develop the first working version. This language was initially called 'OAK', but was renamed "java" in 1995. [6]The primary motivation was the need for a platform independent language that can be used to create embedded software's in a cost effective way.

D. Features of Java[6]:-

- Simple, Small, Familiar.
- Compiled and Interpreted.
- Portable.
- Platform -independent.
- Distributed.
- Dynamic and Extensible.

II. JVM (Java Virtual Machine)

A Java Virtual Machine is a piece of software that is implemented on non virtual hardware and on standard operating system.[17] A JVM provides an environment in which java byte code can be executed ,enabling such features as automatic exception handling, which provides "root-cause" debugging information for every software error (exception),independent of the source code. [10] A JVM is distributed along with a set of standard class libraries that implement the java application programming interface (API). Appropriate API's bundled together form the Java Runtime Environment (JRE). JVM are available for many hardware and software platforms. The use of same byte code for all JVMs on all platforms allows java to be described as a "Compile once, run anywhere" programming language, as opposed to "write once, compile anywhere", which describes cross –platform compiled languages.[9] Thus, the JVM is a crucial component of the Java platform. Java Byte code is an intermediate language which is typically compiled from java, but it can also be compiled from other programming languages. E.g. Ada source code can be compiled to java byte code and execute on a JVM. Byte code verifier, the JVM verifies all byte code before it is executed. [10] This verification consists primarily of three types of checks:

- Branches are always to valid locations.
- Data is always initialized and references are always type safe.
- Access to "private" or "package private" data and method is rigidly controlled.

The first two of these checks take place primarily during the "verification" step that occurs when a class is loaded and made eligible for use. The third is primarily preformed dynamically, when data items or methods of a class are first

accessed by another class. The verifier permits only some byte code sequences in valid program, e.g. a jump (branch) instruction can only target an instruction within the same method.

A. Memory Areas for JVM:[3]

- Method area.
- Class description.
- Constant pool.
- Heap.
- Garbage collection.
- Stack.

B. JVM Architecture:-

The JVM is basically a stack based machine, with a 32 bit word size, using 2's complement arithmetic. JVM is an efficient way of getting memory protection on simple architectures that lack an MMU (Memory Management Unit).[4] [5]This is analogous to managed code in Microsoft's .NET Common Language Runtime.

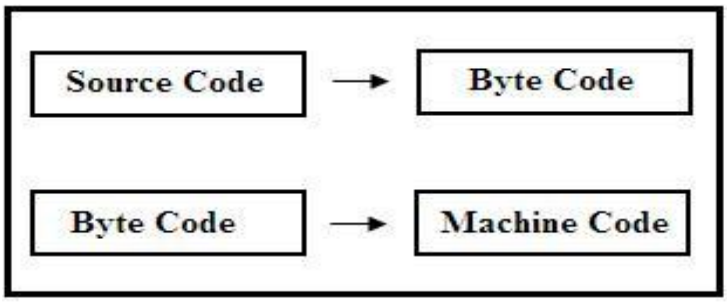

Figure1.1 JAVA VIRTUAL MACHINE.

- Optop: Pointer to the top of the operand stack for the currently active method.
- Frame: Pointer to the stack frame of the currently active method.
- Vars: Pointer to the beginning of the local variables of the currently active method.

The JVM Architecture is very much stack oriented. Most instruction access the stack in some way. [9] The stack is also used for method calls, as with many classical machine architectures. A method call produces a new stack frame, which is pushed onto the stack and return pops the frame from the program's stack. Almost all JVM instructions are stack based. [2] In the JVM specification, the behavior of a virtual machine instance is described in terms of subsystem, memory areas, data types, and instruction. These components describe an abstract inner architecture for the abstract java virtual machine. It is more to provide a way to strictly define the required behavior of implementations.[4] The Specification defines the required behavior of any java virtual machine implementation in term of these abstract components and there interaction.

C. Byte code instructions

The JVM has instructions for the following groups of tasks:

- Load and store
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack management (push / pop)
- Control transfer (branching)
- Method invocation and return
- Throwing exceptions
- Monitor-based concurrency

The aim is binary compatibility. Each particular host operating system needs its own implementation of the JVM and runtime. [3] These JVMs interpret the byte code semantically the same way, but the actual implementation may be different. More complex than just emulating byte code is compatibly and efficiently implementing the Java core API that must be mapped to each host operating system. [9] Figure 1.2 shows a block diagram of the java virtual machine that includes the major subsystems and memory areas described in the specification. Each Virtual machine has a class loader subsystem, also has an execution engine, a mechanism responsible for executing the instructions contained in the methods of loaded classes.

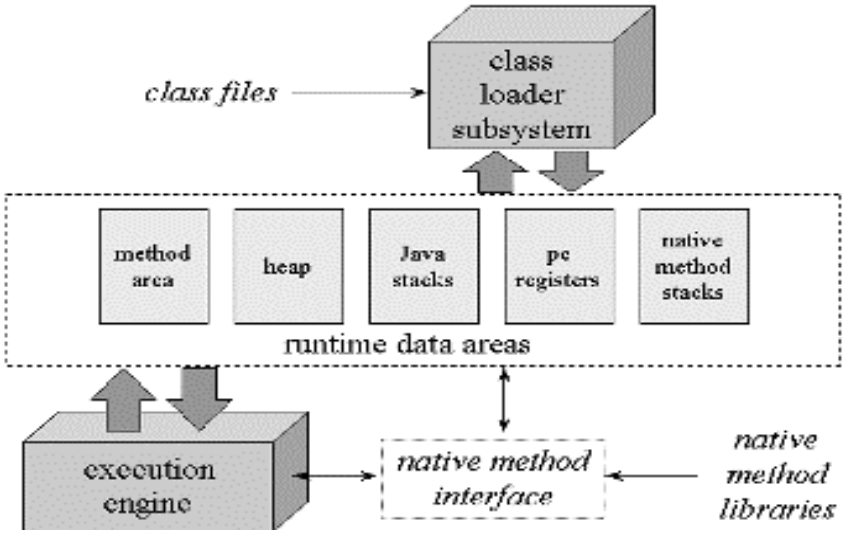

Figure 1.2 Internal Architecture of JVM [9]

Virtual machine architecture allows very fine-grained control over the actions that code within the machine is permitted to take. This is designed to allow safe execution of untrusted code from remote sources, a model used by Java applets. Applets run within a VM incorporated into a user's browser, executing code downloaded from a remote HTTP server. The remote code runs in a restricted "sandbox", which is designed to protect the user from misbehaving or malicious code. Publishers can purchase a certificate with which to digitally sign applets as "safe", giving them permission to ask the user to break out of the sandbox and access the local file system, clipboard or network. At the heart of the Java platform lies the Java Virtual Machine, or JVM. Most programming languages compile source code directly into machine code, suitable for execution on particular microprocessor architecture. The difference with Java is that it uses byte code - a special type of machine code. Java byte code executes on a special type of microprocessor. Strangely enough, there wasn't a hardware implementation of this microprocessor available when Java was first released. Instead, the processor architecture is emulated by what is known as a "virtual machine". [17] This virtual machine is an emulation of a real Java processor - a machine within a machine. The only difference is that the virtual machine isn't running on a CPU - it is being emulated on the CPU of the host machine.

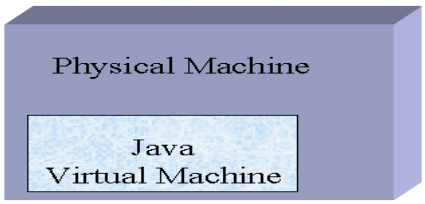

Figure 1.3 JVM emulation run on a physical CPU [10]

The Java Virtual Machine is responsible for interpreting Java byte code, and translating this into actions or operating system calls [8]. For example, a request to establish a socket connection to a remote machine will involve an operating system call. Different operating systems handle sockets in different ways - but the programmer doesn't need to worry about such details. It is the responsibility of the JVM to handle these translations, so that the operating system and CPU architecture on which Java software is running is completely irrelevant to the developer.

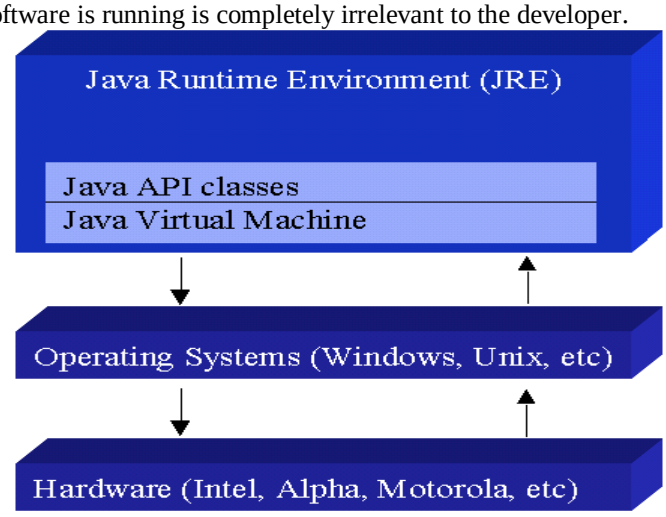

Figure 1.4 - JVM handles translations [10]

The Java Virtual Machine forms part of a large system, the Java Runtime Environment (JRE). Each operating system and CPU architecture requires a different JRE. [5] The portability of Java comes from implementations on a variety of CPUs and architectures. Without an available JRE for a given environment, it is impossible to run Java software. The Java Virtual Machine provides a platform-independent way of executing code, by abstracting the differences between operating systems and CPU architectures. Java Runtime Environments are available for a wide variety of hardware and software combinations, making Java a very portable language.[3]. Thanks to the JVM, the dream of Write Once-Run Anywhere (WORA) software has become a reality.

D. JVM security architecture

Java virtual machine is abstract computers that can load and execute java programs .It contain a virtual processor of byte code language, stack, registers and it interprets about 200 instructions. [2] JVM operation is defined in Java Virtual Machine Specification, which among other also defines:

- Class files format.
- Java byte code language instruction set.
- Byte code verifier behavior.

1) Class Loader:

- Special Java runtime objects that are used for loading Java classes into the Java Virtual Machine.
- They provide JVM with functionality similar to the one of a dynamic linker.
- Each class Loader defines a unique namespace.
- For every class loaded into JVM a reference to its class Loader object is maintained.

a) Class Loader Types:

- System Class Loader.
- Applet Class Loader.
- RMI Class Loader.
- User Defined Loader.

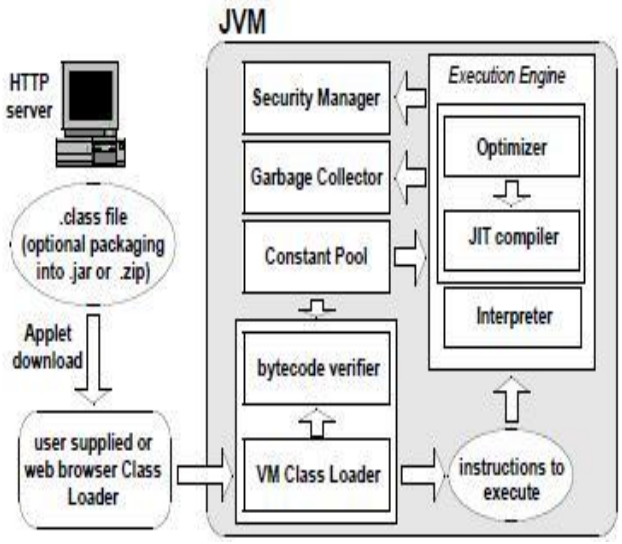

Figure 1.5 JVM security architecture. (Life Cycle of a Java Class File)[2]

3) Security Manager:

- It guards security policies for java applications.
- It is always consulted before any potentially dangerous operation is requested by java application.
- It implements appropriate "check" methods that implement a given security policy.
- It is responsible for enforcing the applet sandbox security restrictions.

4) Byte code verifier:

- It is responsible for verifying that class files loaded to Java Runtime have a proper internal structure and that they are consistent with each other.
- It enforces that java Byte code is type safe.
- Most of its work is done during class loading and linking.
- For every execution path that can occur in a verified code, it checks type compatibility of arguments passed to methods and used as byte code instructions' operands.

III. JNDI (Java Naming and Directory Interface)

The JNDI specification was first released by Sun Microsystems on March 10, 1997.[11]The Java Naming and Directory Interface (JNDI) is part of the Java platform, providing applications based on Java technology with a unified interface to multiple naming and directory services. The Java Naming and Directory Interface (JNDI) is an application programming interface (API) for accessing different kinds of naming and directory services.[13] JNDI is not specific to a particular

naming or directory service, it can be used to access many different kinds of systems including file systems; distributed objects systems like CORBA, Java RMI, and EJB,[14] and directory services like LDAP, Novell NetWare, and NIS+. JNDI is similar to JDBC in that they are both Object-Oriented Java APIs that provide a common abstraction for accessing services from different vendors. While JDBC can be used to access a variety of relational databases, JNDI can be used to access a variety of naming and directory services.[12] Using one API to access many different brands of a service is possible because both JDBC and JNDI subscribe to the same architectural tenet: Define a common abstraction that most vendors can implement. The common abstraction is the API. It provides an objectified view of a service while hiding the details specific to any brand of service. The implementation is provided by the vendor, it plugs into the API and implements code specific to accessing that vendor's product. JNDI provides two APIs and one SPI. JNDI has a naming API that allows Java applications to access naming systems like CORBA's Naming services and a directory API that extends the naming service to provide access to directory services like LDAP. JNDI also has a SPI (Service-Provider Interface) which is a programming model that vendors use to write JNDI plug-ins or implementations for their specific product. Each vendor's plug-in is called a service-provider. A service-provider implements the JNDI APIs so that a Java application can access that vendor's product. For the most part, JNDI hides the implementation details of the a service-provider so that Java developer that uses JNDI can use the same objects and method regardless of the brand of naming or directory service accessed. This is the real power behind APIs like JDBC and JNDI.[15] They provide one programming model for accessing many different products; there is no need to learn a different programming model every time a different product is used. JNDI is naming service that maintains a set of bindings in the server container. Bindings relate names to objects. All objects in a naming system are named in the same way (that is, they subscribe to the same naming convention). Clients use the naming service to locate objects by name. These JNDIs are used in to get the Data source, EJBs, JMS, Java Mail, LDAP, etc, these services should be defined as names in server and from client you need to call them in order to access appropriate services, these names should be unique.

A. Naming System:

Naming System	Component Separator	Names
UNIX™ file system	"/"	/usr/hello
DNS	"."	sales.Wiz.COM
LDAP	"," and "="	cn=Rosanna Lee, o=Sun, c=US

The UNIX™ file system's naming convention is that a file is named from its path relative to the root of the file system, with each component in the path separated from left to right using the forward slash character ("/"). The UNIX pathname, /usr/hello, for example, names a file hello in the file directory usr, which is located in the root of the file system. DNS naming convention calls for components in the DNS name to be ordered from right to left and delimited by the dot character ("."). Thus the DNS name sales.Wiz.COM names a DNS entry with the name sales, relative to the DNS entry Wiz.COM. The DNS entry Wiz.COM, in turn, names an entry with the name Wiz in the COM entry. The Lightweight Directory Access Protocol (LDAP) naming convention orders components from right to left, delimited by the comma character (","). Thus the LDAP name cn=Rosanna Lee, o=Sun, c=US names an LDAP entry cn=Rosanna Lee, relative to the entry o=Sun, which in turn, is relative to c=us.[15] LDAP has the further rule that each component of the name must be a name/value pair with the name and value separated by an equal's character ("=").

B. Directory Concepts

Many naming services are extended with a directory service. A directory service associates names with objects and also associates such objects with attributes.*[14]* Directory service = naming service + objects containing attributes

1) Directories and Directory Services

A directory is a connected set of directory objects. A directory service is a service that provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory. The service is accessed through its own interface. Many examples of directory services are possible:

- Network Information Service (NIS)- NIS is a directory service available on the UNIX operating system for storing system-related information, such as that relating to machines, networks, printers, and users.
- Sun Java Directory Server
  The Sun Java Directory Server is a general-purpose directory service based on the Internet standard LDAP
- Novell Directory Service (NDS)
  NDS is a directory service from Novell that provides information about many networking services, such as the file and print services.

JNDI is included in the Java SE Platform. To use the JNDI, you must have the JNDI classes and one or more service providers.[15] The JDK includes service providers for the following naming/directory services:

- Lightweight Directory Access Protocol(LDAP)
- Common Object Request Broker Architecture (CORBA)
- CommonObjectServices (COS) name service
- Java Remote Method Invocation (RMI) Registry.
- Domain Name Service (DNS)

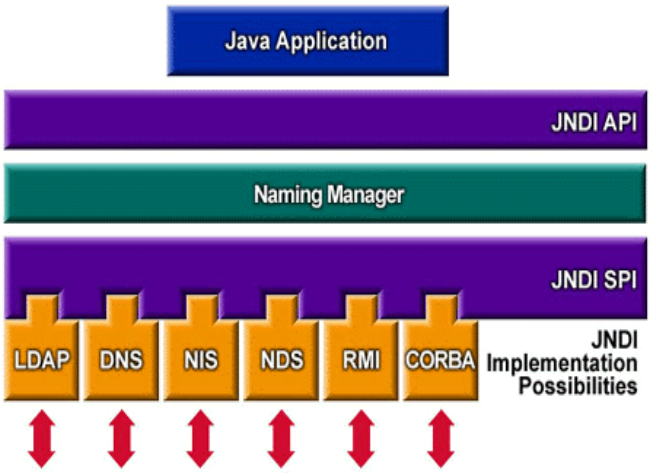

Figure 2.1 Services of JNDI.[15]

C. Similarities between JDBC and JNDI

The Java Naming and Directory Interface (JNDI) is for LDAP programming, but Java Database Connectivity (JDBC) is for SQL programming. There are several similarities between JDBC and JNDI/LDAP (Java LDAP). Despite being two completely different APIs with different pros and cons, they share a number of less flattering characteristics:

- They require extensive plumbing code, even to perform the simplest of tasks.
- All resources need to be correctly closed, no matter what happens.
- Exception handling is difficult.

The above points often lead to massive code duplication in common usages of the APIs. As we all know, code duplication is one of the worst code smells. All in all, it boils down to this: JDBC and LDAP programming in Java are both incredibly dull and repetitive. Spring JDBC, a part of the spring framework, provides excellent utilities for simplifying SQL programming. We need a similar framework for Java LDAP programming.

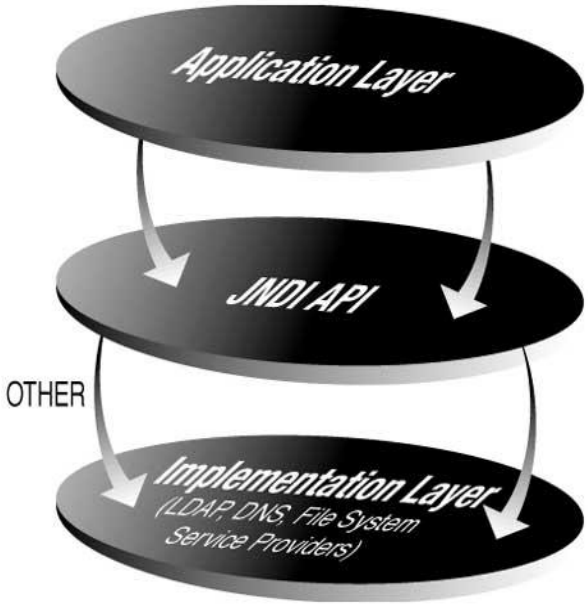

Figure 2.2 Working of JNDI with Layers. [16]

IV. COMPARISION OF JVM WITH OTHER VIRTUAL MACHINE.

A. *Various Virtual Machines of different languages*

- **LLVM**:- The Low Level Virtual Machine (LLVM) is a compiler infrastructure written in C++ that is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages.[18] Originally implemented for C/C++, the language-agnostic design (and the success) of LLVM has since spawned a wide variety of front ends, including Objective C, Fortran, Ada, Java byte code etc.
- **Mono**: - It is a software platform designed to allow developers to easily create cross platform applications. It is an open source implementation of Microsoft's .Net Framework based on the ECMA standards for C# and the Common Language Runtime.[18] We feel that by embracing a successful, standardized software platform, we can lower the barriers to producing great applications for Linux.

- **Zend Engine: -** Zend Engine is used internally by PHP as a complier and runtime engine. PHP Scripts are loaded into memory and compiled into Zend opcodes. These opcodes are executed and the HTML generated is sent to the client..

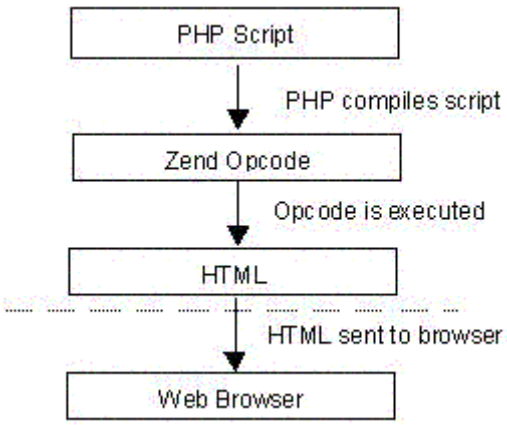

Figure 2.3 PHP Zend Engine

It is a register based interpreter. There is a stack, but that's used for calling functions only. [18] The operands to the opcodes are pointed to by the opcodes in the case of compiled variables, or in symbol tables otherwise. That's as close to a register machine as we can get I think, but it's not very close to a stack machine. In a stack-based VM, the operands to an opcode would be implicit, with add for example using the top two stack operands, and that's not the case at all.

TABLE 1. Comparison of Various Virtual Machines.

Virtual machine	Machine model	Memory management	Code security	Inter Preter	JIT	Pre compilation	Shared libraries	Common Language Object Model	Dynamic typing
JVM[17]	stack	automatic	Y	Y	Y	Y	Y	Y	Y
LLVM	register	manual	N	Y	Y	Y	Y	Y	N
Mono	stack	automatic or manual	Y	Y	Y	Y	Y	Y	Y
Zend Engine	register	automatic	N	Y	N	source to byte code	Y	N	Y

- Virtual machine instructions process data in local variables using a main model of computation, typically that of a stack machine, register machine, or random access machine often called the memory machine. Use of these three techniques is motivated by different tradeoffs in virtual machines vs physical machines, such as ease of interpretation, compilation, and verifiability for security.[18]
- Memory management in these portable virtual machines is addressed at a higher level of abstraction than in physical machines. Some virtual machines, such as the popular JVM, are involved with addresses in such a way as to require safe automatic memory management by allowing the virtual machine to trace pointer references, and disallow machine instructions from manually constructing pointers to memory. Other virtual machines, such as LLVM, are more like traditional physical machines, allowing direct use and manipulation of pointers. CIL offers a hybrid in between, offering both controlled use of memory (like the JVM, which allows safe automatic memory management), while also offering an 'unsafe' mode that allows direct manipulation of pointers in ways that can violate type boundaries and permission.
- Code security generally refers to the ability of the portable virtual machine to run code while only offering it a prescribed set of capabilities. For example, the virtual machine might only allow the code access to a certain set of functions or data. The same controls over pointers which make automatic memory management possible and allow the virtual machine to ensure type-safe data access are used to assure that a code fragment is only allowed to certain elements of memory and cannot sidestep the virtual machine itself. Other security mechanisms are then layered on top as code verifiers, stack verifiers, and other techniques.
- An interpreter allows programs made of virtual instructions to be loaded and immediately run without a potentially costly compilation into native machine instructions. Any virtual machine which can be run can be interpreted, so the

column designation here refers to whether the design includes provisions for efficient interpretation (for common usage).

- Just-in-time compilation or JIT refers to a method of compiling to native instructions at the latest possible time, usually immediately before or during the running of the program.[1] The challenge of JIT is more one of implementation than of virtual machine design, however, modern designs have begun to make considerations to help efficiency. The simplest JIT techniques simply perform compilation to a code-fragment similar to an offline compiler. However, more complicated techniques are often employed, which specialize compiled code-fragments to parameters that are known only at runtime (see Adaptive optimization).
- Pre-compiling refers to the more classical technique of using an offline compiler to generate a set of native instructions which do not change during the runtime of the program. Because aggressive compilation and optimization can take time, a precompiled program may launch faster than one which relies on JIT[18] alone for execution. JVM implementations have mitigated this startup cost by using interpretation initially to speed launch times, until native code-fragments can be generated through JIT.
- Shared libraries are a facility to reuse segments of native code across multiple running programs. In modern operating systems, this generally means using virtual memory to share the memory pages containing a shared library across different processes which are protected from each other via memory protection.[18] It is interesting that aggressive JIT techniques such as adaptive optimization often produce code-fragments unsuitable for sharing across processes or successive runs of the program, requiring a tradeoff be made between the efficiencies of precompiled and shared code and the advantages of adaptively specialized code. For example, several design provisions of CIL are present to allow for efficient shared libraries, possibly at the cost of more specialized JIT code. The JVM implementation on Mac OS X uses a Java Shared Archive (apple docs) to provide some of the benefits of shared libraries.

TABLE 1.1 Implementation Areas of Various Virtual Machines.

Virtual machine	Languages	Comments	Implementation Language	SLOC
JVM	Java, Jython, C, C++, and several others	Reference implementation by Sun ; OpenJDK: code under GPL ;	JDK, OpenJDK JIT : Java, C	JVM is around 6500k lines[18]
LLVM	C, C++, Objective-C, Ada, and Fortran	C and C++ output are supported. bytecode is named "LLVM Bytecode (.bc)". Assembly is named "LLVM Assembly Language (*.ll)".	C++	430 k
Mono	CLI languages including: C#, VB.NET, and others	clone of Common Language Runtime.	C#, C	2332k
Zend Engine	C	Compiler and runtime engine for PHP	PHP	75k

V. CONCLUSION

According to the study related to comparison of virtual machine shows that every virtual machine provides there individual roles in the specific languages. Given comparison shows that JVM provides the better way to safe memory management system automatically as compare to other virtual machines. It also provides better code security, i.e type safe. JVM provide us the facility of portable code, which support the mechanism of WORA (Write Once, Run Anywhere).JVM is Stack based Architecture, so the operation like push and pop takes place.

ACKNOWLEDGMENT

The work described in this paper was supported by School of Computer Science & Engineering , Bahra University Shimla Hills, India.

References

[1] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Ko-matsu, and T. Nakatani, *"Overview of the IBM Java Just-in-Time Compiler."* IBM Systems Journal, 39(1):175-193, 2000

[2] The Last Stage of Delirium Research Group ,Poland, *"Java and Java Virtual Machine security vulnerabilities and their exploitation techniques."*, October 3rd - 4th 2002.

[3] J. Meyer and T. Downing., "*Java Virtual Machine."* O'Reilly, 1997.

[4] James Gosling, Bill Joy, Guy Steele, and GiladBracha,*"The Java Language Specification*" , Addison-Wesley, 2000, JavaSpec.
[5] Tim Lindholmand Frank Yellin. *"The Java Virtual Machine Specification",* Addison-Wesley, 1999, JVMSpec.
[6] E. Balaguruswami, *"Programming with java",* Tata McGraw-Hill
[7] http://www.java.sun.com/docs/books/vmspec/
[8] http://www.bitpipe.com/tlist/Java-Virtual-Machine.html
[9] http://www.artima.com/insidejvm/ed2/jvm2.html
[10] http://www.javacoffeebreak.com/articles/inside_java/insidejava-jan99.html
[11] http://java.sun.com/javase/technologies/core/jndi/index.jsp
[12] http://en.wikipedia.org/wiki/Java_Naming_and Directory Interface
[13] http://www.javasoft.com/products/jndi
[14] http://java.sys-con.com/node/36454
[15] http://java.sun.com/developer/technicalArticles/Programming/jndi/index.html
[16] http://gemsres.com/photos/story/res/36454/fig1.jpg
[17] http://en.wikipedia.org/wiki/Java_Virtual_Machine
[18] http://en.wikipedia.org/wiki/Comparison_of_application_virtual_machines

