Step Up to Modern Cloud Development

GraalVM: Language-Level Virtualization

Universal language runtime revolutionizes the ability to 'write once, run anywhere.'

Vol 1.0, April 2018

GraalVM™

ORACLE®

Introduction

Computer virtualization has been a consistent trend for the past 20 years. While "virtualization" and "virtual machines" (VMs) mean different things in different contexts, they have a couple of goals in common: the isolation of code from other code running in the same machine and the ability to "write once, run anywhere."

To date, this virtualization vision has been incomplete. That's because traditional virtualization approaches don't address the issue of multiple programming languages.

Meanwhile, the number of programming languages in use continues to increase, with the most popular language (Java) garnering only about 10 percent market share. So virtualization's multilingual problem is growing.

Attempts to build multilingual runtimes to fill this void have fallen short with poor performance and an inability to support all the semantics, features, and native extensions libraries of the new languages. But all that is changing with GraalVM language-level virtualization.

Current Virtualization Approaches

As mentioned, "virtualization" can mean different things, depending on context. For example, a Java VM is very different than a VM like VirtualBox or Xen. In the case of the Java VM, the goal is to run your program on any kind of processor without modification. We'll call this processor virtualization. In the case of a VM like VirtualBox or Xen, the goal is to run your program on any operating system. So we'll call this OS virtualization. Both kinds of VMs provide some level of isolation and safety to keep the programs they execute from misusing the resources of the underlying hardware in a "sandbox." They allow multiple tenant applications to share the hardware for efficiency reasons.

One recent trend in the world of virtualization is to move towards lighter sandboxes for tenant applications. A major step has been to move from VMs that provision a separate OS per tenant to a "container" that isolates applications but shares a single underlying OS among multiple tenants. This allows a sandbox to be smaller, and more can fit on a single server. A second phase towards smaller sandboxes is called "serverless computing," also called functions-as-a-service, or FaaS. In serverless computing, a container is allocated to an application only when it is being used, and cloud customers are only billed for the time used. If usage of an application is sporadic, serverless computing can be even more efficient than containers, since a container is spun up only when needed. Of course, if the time to start up a container is high, then serverless computing is less efficient.

Multilingual Challenges

As mentioned, neither OS virtualization nor processor virtualization are a complete solution to enabling "write once, run anywhere." They don't address the growing number of programming languages in use (Figure 1).

Generally, a library is unavailable for use in an application if it is written in a different language than the consuming application. An exception is when the library can tolerate high overhead per call and operations don't mind managing multiple language runtimes with different configuration and resource management.[1] If you write a library in Python, for example, there will be much more overhead to call that library from a Java application than from a Python application.

Expense and development burdens

Developing an application in multiple languages is also more expensive, as there is no common set of tools for debugging and profiling multilingual applications. Even more troublesome, any data needed by more than one language in your application must be copied into each language runtime and kept in sync across the various languages.

Many organizations try to enforce a standard language for applications to reduce runtime inefficiencies and avoid the additional costs of multilingual development. However, this is not easy. First of all, each developer is used to writing code in a particular language or two and feels most productive in a particular language. Hiring programmers only with experience in a particular language can often make it much harder to find staff.

It's also common in large organizations for multilingualism to creeps in with acquired companies, which have often made different programming language choices. Usually, it is impossible to force all of the company's technology onto one language platform.

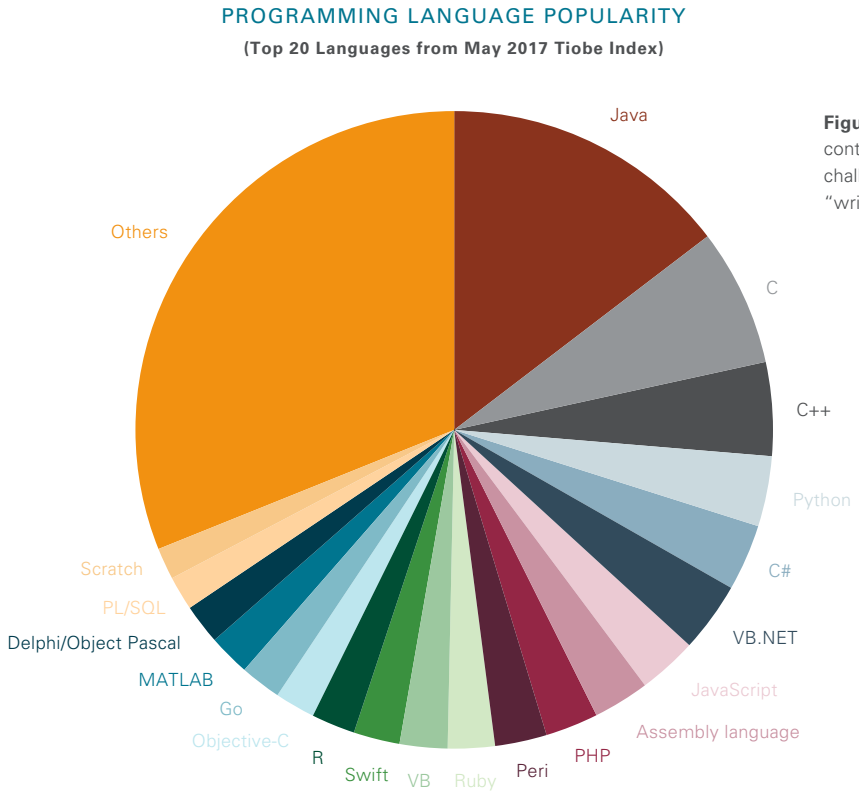

Figure 1: Programming languages continue to grow in number, challenging virtualization's goal to "write once, run anywhere."

1 *As an aside, most popular languages today (Java, Python, Ruby, R, JavaScript) involve an application runtime that must be executed, and which in turn executes the program (sometimes called "high level languages"). "Native languages" are programming languages that compile to an executable directly accessing the OS, and don't need a runtime, such as C, C++, Go, and FORTRAN.*

However, every company, large and small, faces the reality that different languages are better at different jobs, in part because of the community and ecosystem that have grown up around those languages. For example, Ruby is used frequently for web front ends and fancy user interfaces, and you are more likely to find a library to do something like manipulate a PNG image in Ruby than in JavaScript, Python, or other languages. Python is used frequently for data science and machine learning, and there are more ML libraries available in Python than in other languages. R is popular among statisticians, and the latest statistical techniques are most likely found in an R library.

Lack of cross-engine runtime embeddability

Another limit is attempting to use a library in a runtime engine like a database or a web server. Those engines often have limits on what languages they can use to extend or customize their functionality. Relational databases frequently offer only a language like SQL's procedural extensions (e.g., Oracle's PL/SQL or Microsoft's Transact-SQL). A web server like NGINX that is used for something like a load balancer cannot run arbitrary language code to help decide where incoming requests should be routed. The primary issue is that language runtimes often want to take control of managing resources like memory or processor threads, which databases and other engines want to control. Language runtimes are often not embeddable into other engines, limiting "write once, run anywhere." Note, though, that some newer databases, especially in the cloud, are taking the initiative to try to embed a particular language runtime of their choice because of the need for extensibility.

One exception to language interoperability is that most language runtimes do provide a means to call out to native code. They do this to provide more direct access to the operating system and, more frequently, for efficiency, since native code is often faster than higher-level languages. You can write a library that is usable for many languages by writing it in native code. One downside to doing this is that calls between the higher-level language and native code are usually clunky to write and involve significant performance overhead. Each language has a native call interface (e.g., the Java Native Interface, or JNI) that is used to make those cross-language calls. Making calls via native interfaces involves memory allocation and type conversion, since the language runtimes have their own memory management and type system (their data must have a certain memory layout). A second downside to writing native libraries, in addition to lower levels of developer productivity, is that these native call interfaces are a common source of bugs and security vulnerabilities. The reason is that the native code is responsible for maintaining all of the semantics of the language runtime, such as object lifetimes.

Overhead Per VM Sandbox Remains High

The virtualization world is moving to lightweight containers like Docker, allowing more isolated application instances per server, since each container doesn't contain the OS. So the amount of memory needed to run the container efficiently, called the Resident Set Size, or RSS, is much lower, allowing more containers to fit into a server with a given amount of physical memory. A lighter container doesn't make a huge difference in the number of CPU cycles needed to run each application, given that the OS will do the same work whether or not it is shared by multiple containers. Still, most enterprise servers today incur significantly more expense from provisioning DRAM than they do from the CPUs. In addition, most general-purpose applications[2] in the data center today are limited more by memory bandwidth than by CPU cycles. So, memory optimization is the most important consideration over CPU optimization, and reducing the size of containers makes sense.

However, after removing the OS per container, there is usually a need to run a language runtime per Docker container, and those runtimes require a lot of memory overhead. While a native application to print "Hello World" needs only around 500 kB, languages like Java or JavaScript need around 20MB in overhead for the most trivial program. 20MB can be significantly more overhead than the application itself needs, especially when running small programs, or "microservices," in the container. That means fewer containers can fit in a server than what the application needs, due to the language runtime overhead. Not only do the language runtimes carry substantial memory overhead, many of them need significant work from the CPU to start up, which has an impact on the value of serverless infrastructure (Figure 2).

One technique that was popular in the past for running applications with low overhead, particularly in Java, was to use an application server. This approach allowed multiple Java applications to share a single Java language runtime, which can amortize the runtime's cost across many applications. Unfortunately, application servers provided an insufficient level of isolation between the various tenants; sharing the VM in Java means sharing the same heap and garbage collector, and that allows one memory-intensive tenant to make the other tenants much slower. In addition, a Java application server cannot provide good isolation when the tenants are calling libraries in native code, because the native code from the tenants will share the same address space.

Figure 2: Overhead to Run "Hello World" in various language runtimes

LANGUAGE	VIRTUAL MACHINE	INSTRUCTIONS	TIME	MEMORY	
C helloworld		100,000	< 10 ms	450 KByte	printf("Hello World!\n")
GNU helloworld 2.10		300,000	< 10 ms	800 KByte	C with argument parsing
JavaScript	V8	10,000,000	< = 10 ms	18,000 KByte	version 5.6.0
JavaScript	Spidermonkey	77,000,000	20 - 30 ms	10,000 KByte	version C52.0a1
Java	Java Hotspot VM	140,000,000	40 ms	24,000 KByte	JDK 8 update 111
JavaScript	GraalVM in Hotspot (JDKB)	N/A	650 ms	120,000 KByte	GraalVM release 0.19

2 By "general purpose," I mean an application that does a variety of work such as a user interface, data manipulation, and business logic, in contrast to a specialized application like a machine learning workload, which is very CPU-intensive.

Language-Level Virtualization with GraalVM

Oracle Labs has been developing a third kind of technology we call "language-level virtualization" as a part of the GraalVM project. What GraalVM does is provide a universal language runtime that can run any language. Whereas a conventional language runtime is designed for a specific language, GraalVM is an additional level of "meta" that runs things that run languages. There have been other attempts to build multilingual runtimes, such as the Microsoft Common Language Runtime (CLR) and attempts to host other languages on the Java VM. However, those efforts have suffered from three issues in running languages they weren't designed for:

- An inability to support all of the semantics and features of the new languages
- An inability to support all of the native extensions libraries of the new language ecosystem
- Poor performance on the languages that the runtime wasn't designed for ahead of time

GraalVM doesn't suffer from any of these issues, because it starts with language fundamentals.

100 times faster

The most basic way to develop a program language runtime is to build an interpreter—a program that takes each line of the application code, parses it, and branches it to a specific subroutine for each kind of operator or expression in that language. An interpreter is easy to build, but quite slow, since it is generally more work to parse the application code to figure out what to do, than it is to just do the work. For example, an interpreter for a + b must first separate out the variables a and b from the "plus" operator, figure out if a and b are strings or numbers, and then call the specific function for that kind of "plus" operator. What GraalVM does is to take an interpreter, written to a specific Java API called "Truffle," and automatically convert it to a compiler. This technique makes interpreters ~100 times faster by automatically deriving high-performance machine code and removing any interpretation overhead.

The idea of automatically converting language interpreters into compilers has been around since the 1970s, when it was first published by Yoshihiko Futamura as the "Futamura projection." However, the Futamura projection was impractical because it didn't generate a compiler that was as good as one that was built by hand for the particular language. What Oracle Labs has finally shown is how to make a technique for the Futamura projection practical – even when dealing with dynamic languages with complex semantics. The theory behind how GraalVM creates high-quality compilers using "partial evaluation" was published in 2017 at the preeminent Programming Language Design & Implementation (PLDI) academic conference on programming languages in a paper entitled "Practical Partial Evaluation for High Performance Dynamic Runtimes." GraalVM watches the behavior of each interpreter to "learn" the semantics of its language and then incrementally compiles the parts of the application code that are frequently used (or "hot").

Cross-language calls with zero overhead

You can use as many interpreters for as many languages as you want in a GraalVM runtime, writing them all to the Truffle API. Because there is not much difference to GraalVM between two Truffle languages, GraalVM can call across language boundaries with zero overhead. GraalVM can even do a compiler optimization called "inlining" across languages, treating function calls as if they were part of the code calling the function to eliminate overhead. An important Truffle feature is to provide "logical / physical data layout independence," which means that any memory layout can be used for objects in each language interpreter. In fact, a single language may have multiple different ways to lay out an object in memory. This is an important performance optimization; for example, data from the network doesn't have to be copied out of network buffers into a language object.

An even more interesting effect is that objects from foreign programming languages can be used by other GraalVM languages and treated like objects in the current language. This allows intermixing languages together at a very fine-grained level without copying data.

Language interpreter for native code

The other key technology in GraalVM is a special language interpreter that handles native code. Most native languages (C, C++, FORTRAN, Rust, COBOL, and Go) are supported by an open-source compiler called a Low-Level Virtual Machine (LLVM). LLVM has a very useful feature for GraalVM, which is that the LLVM compiler can generate an intermediate language called "bitcode" from all of its supported languages. Bitcode is fairly low-level and is best understood as a kind of portable Assembly language. GraalVM has an interpreter for that bitcode, and GraalVM can then compile that bitcode into machine code like a conventional compiler. The GraalVM interpreter for LLVM bitcode allows the native extensions for other language interpreters to run in the same GraalVM tenant, keeping the native code isolated from other tenants' data (Figure 3).

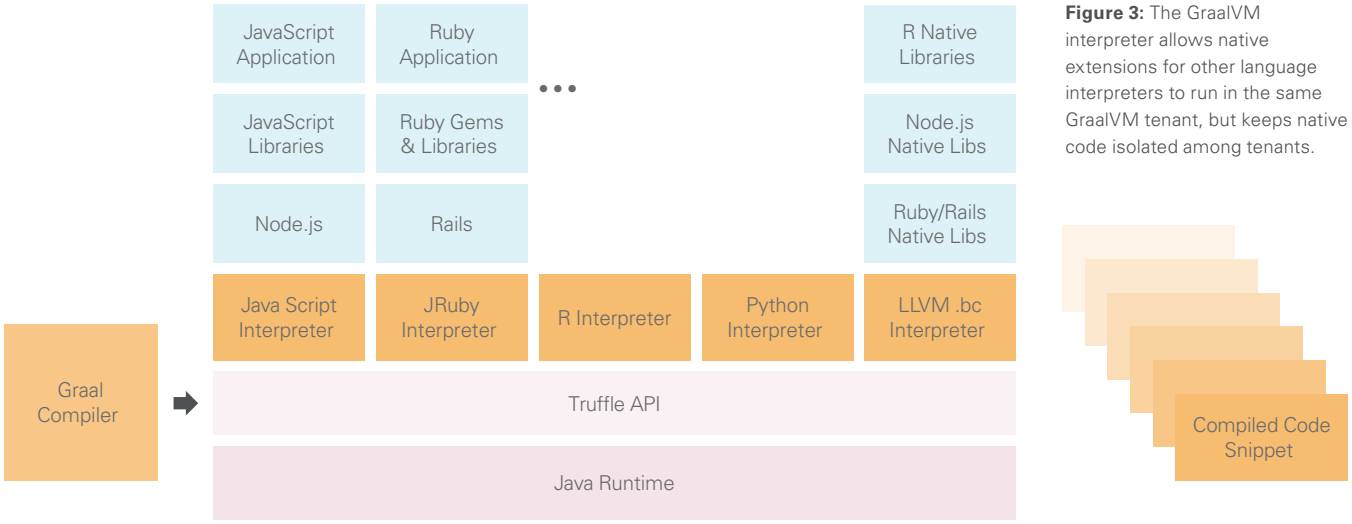

Figure 3: The GraalVM interpreter allows native extensions for other language interpreters to run in the same GraalVM tenant, but keeps native code isolated among tenants.

What Code Should I Trust?

Most new code today is being written in a dynamic language (e.g., Java, JavaScript, R, Ruby, or Python), where the program is compiled at runtime using a "just-in-time" (JIT) compiler.[3] The JIT compiler watches what the program is doing for a while, records the activity in a "profile," and then optimizes its compilation for that profile. Dynamic languages need a runtime to do the profiling and the JIT compiling and to handle tasks such as memory management.

Code that is performance-critical is often written in static languages, such as C/C++, FORTRAN, Go, and Rust, where the program is usually compiled ahead-of-time (AOT) by a compiler that is separate from the runtime system. The compiler creates a native binary program that can be directly executed by the computer. Static language binaries generally have faster startup time, since they don't have to compile anything when they run. They also have lower overhead (as you can see in the HelloWorld chart in Figure 2). AOT-compiled code is easier to work with for somebody optimizing code manually, as the compiler won't change your code while the program is running. A disadvantage compared to managed languages is that an additional class of security bugs like buffer overflows are relevant for static AOT-compiled languages.

GraalVM gives developers more choices on what code to compile AOT: Java code as well as native code can be compiled AOT, and the static language (native language) code can be compiled dynamically via the LLVM bitcode interpreter. Java code compiled AOT with GraalVM still uses garbage-collected memory with bounds-checks on memory accesses to guarantee memory safety. GraalVM provides a runtime library and a set of tools for building Java AOT called SubstrateVM. Any GraalVM AOT code can be debugged with native tools and can directly call into other native libraries not compiled by GraalVM. Using SubstrateVM allows GraalVM to be embedded in other native runtimes, such as a database. It also provides ways to restrict the portions of the AOT-compiled code that are available to the dynamic language code, using a whitelist, for security reasons. GraalVM was designed to be embeddable and use the underlying system (e.g., the database) tools for security, resource management, and work scheduling.

Using SubstrateVM, we can see substantial benefits on memory overhead and startup time, as shown in the expanded table in Figure 4.

Figure 4: SubstrateVM delivers overhead and startup time benefits

LANGUAGE	VIRTUAL MACHINE	INSTRUCTIONS	TIME	MEMORY	
C helloworld		100,000	< 10 ms	450 KByte	printf("Hello World!\n")
JavaScript	Standalone GraalVM	220,000	< 10 ms	850 KByte	GraalVM release 0.19
GNU helloworld 2.10		300,000	< 10 ms	800 KByte	C with argument parsing
JavaScript	V8	10,000,000	< = 10 ms	18,000 KByte	version 5.6.0
JavaScript	Spidermonkey	77,000,000	20 - 30 ms	10,000 KByte	version C52.0a1
Java	Java Hotspot VM	140,000,000	40 ms	24,000 KByte	JDK 8 update 111

3 For simplicity, any place we discuss the "Java language" can be read as "any language designed to work on the JavaVM", including Java, Scala & Kotlin.

The interesting aspect about running GraalVM either in Java Hotspot or SubstrateVM is that only the precompiled code has access to all of the data in the VM. Therefore the precompiled code must be "trusted" (where the code operates with no security restrictions other than those from the OS). In particular, the language implementations in GraalVM are not trusted code, and only the JIT compiler and the SubstrateVM libraries have to be trusted. This is in contrast to most other VMs, where the entire language application has access to all of the process memory. Limiting the trusted code base limits the code that must be manually analyzed for security vulnerabilities.

The relationships between AOT-code and dynamic code in GraalVM are illustrated in Figure 5.

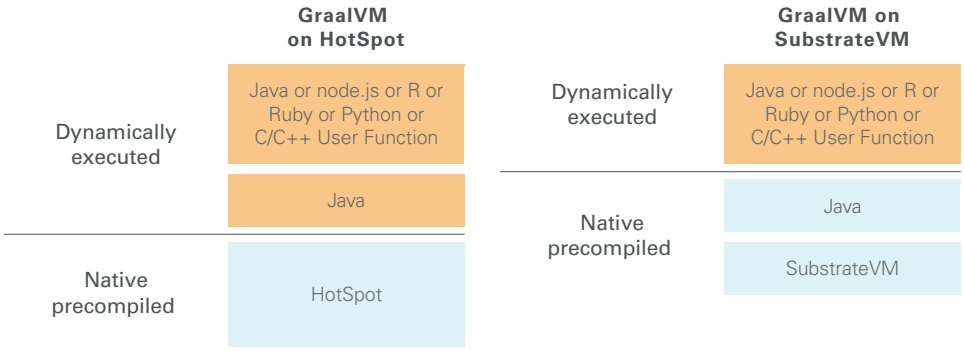

Figure 5: Limiting the trusted code base limits the code that must be manually analyzed for security vulnerabilities.

What about Performance?

In the past, experts believed that a language VM had to be designed and optimized for a single language in order to achieve the best performance. GraalVM demonstrates comparable or better performance across all supported languages as any other implementation. The most notable example demonstrating this is at Twitter, which has published performance results 23% faster using GraalVM. Twitter is confident enough in GraalVM performance that it is using GraalVM in production for their main tweet service to save costs. For languages without industrial investment such as Ruby, R and Python, GraalVM can run code up to 10 times faster.

GraalVM has also demonstrated superior performance inside databases like the Oracle RDBMS. Stored procedures and user-defined functions (UDFs) written in GraalVM and operating on SQL datatypes generally outperform those written in PL/SQL, which was designed explicitly to work with SQL data. In fact, compilation with GraalVM can often outperform built-in SQL functions in particular in cases when arithmetic expressions or other compute-intensive work is in the query.

Conclusion

Language-level virtualization increases developer productivity by allowing developers to use the best language for each task. Libraries of different languages can be used together and there is no need for any overhead when combining programs of different languages. With GraalVM, we demonstrate that one virtual machine can support a large set of diverse programming languages with high performance for each individual language. It can be embedded in data stores and provides lower overhead options for running code in containers, allowing more to fit in a server, and reducing operational costs.

While GraalVM is still a new technology, it is now proven in enough scenarios to be viable for real applications. GraalVM is now delivering the next logical step in virtualization: **write once in any language, run it anywhere in any engine**.

Find out more at http://www.graalvm.org or on Twitter at #graalvm.