OpenVMS Technical Journal V10

Java and OpenVMS: Myths and realities

Jean-Yves Bourlès and Thierry Uso, Consultants

Overview

This article discusses certain myths regarding the use of Java™ programs in the OpenVMS environment. We have identified three myths that, although dispelled by the facts, remain prominent within the OpenVMS and Java communities:

- Java is slow.
- Java is poorly adapted to OpenVMS.
- Java is totally portable.

Myth 1: Java is slow.

The presumed performance problems of Java programs are often used as an excuse for not deploying Java applications in the OpenVMS environment.

Java is neither a compiled nor an interpreted language, in the usual sense of the terms. Rather, Java is a virtual machine (VM) based language. The compilation of Java source code results in a bytecode per Java class, and the bytecodes are platform independent (processor type and OS). The VM interprets the bytecodes by translating them on demand into machine instructions. The principle of the VM means that a Java-written program generally runs slower than one written in a compiled language (C, Ada, and so on) but runs faster than one written in interpreted language (PHP, Perl, and so on). Confirmation of this can be found in the microbenchmark results published on the Computer Language Shootout Benchmarks website (see Figures 1 and 2). The benchmark environment was AMD Sempron, Debian unstable Kernel 2.6.8-1-k7, Java HotSpot 1.4.2_05-b04, C gcc 4.0.3, PHP 5.1.2-1. However, we obtained similar results with Intel® Itanium® rx1620, OpenVMS Version 8.3, Java HotSpot 1.5-2, HP C 7.1-11.

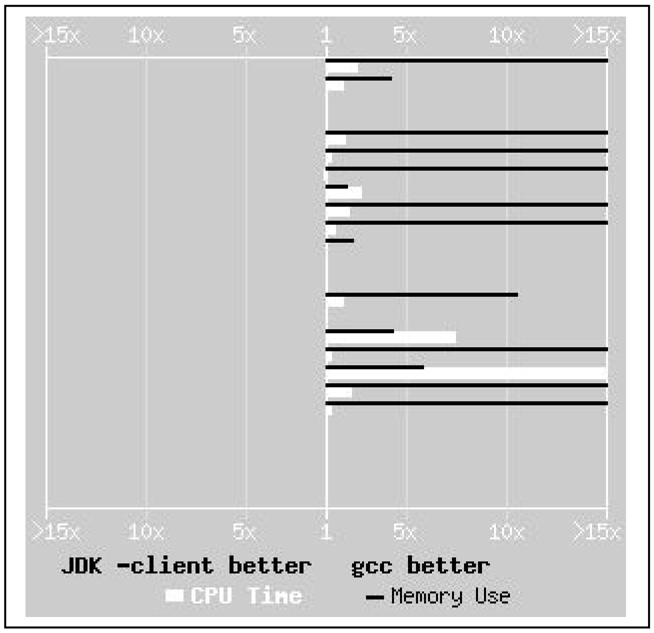

Figure 1: Performance of Java program compared with C program

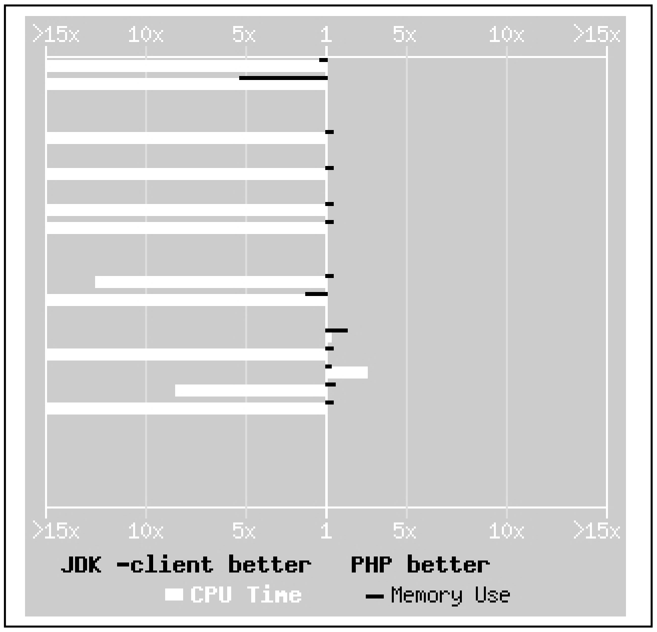

Figure 2: Performance of Java program compared with PHP program

These results show that the progress made in the conception of Java VMs through the introduction of the JIT (Just in Time) technology greatly reduces the performance divide between Java and compiled languages. On the whole, the quality of the written code has a greater impact on performance than does the choice of Java or a compiled language.

Failure to deploy Java applications in an OpenVMS environment can sometimes be justified by the VM-related increased memory consumption, but is increasingly less justifiable for the reason that the language is "slow."

Myth 2 : Java is poorly adapted to OpenVMS.

Launching certain Java applications is very slow on OpenVMS systems. For example, launching the Jetty application server (with the deployment of the web application `Javadoc` from the javadoc.war archive) on a DS20 2x500MHz with 1GB of RAM takes 105 seconds on OpenVMS Version 7.3-2 (Fast VM 1.4.2-3), as compared with 11 seconds on Tru64 UNIX® Version 5.1B (Fast VM 1.4.2-4).

Exactly how poorly adapted to OpenVMS is Java? To answer this question and to identify the reasons for this slowness, we carried out several benchmarks on the platform described in the preceding paragraph.

Benchmark 1 was based on a program that calculated binary search trees; the program is an adapted version of the `Binarytrees` program found on the Computer Language Shootout Benchmarks website. For a depth of 18, the results were as follows:

Operating System	**Delay (seconds)**
OpenVMS	37.4
Tru64 UNIX	27.7

Benchmark 2 was based on a program that sequentially read a file, byte by byte, using the `InputStream()` function. For files of size 10, 100, and 1000 KB, the results were as follows:

Operating System	**Delay 10KB (seconds)**	**Delay 100KB (seconds)**	**Delay 1000KB (seconds)**
OpenVMS	5.0	34.1	332.7
Tru64 UNIX	8.0	79.0	808.4

Benchmark 3 was based on a program that sequentially read a file, block by block, using the `BufferedInputStream()`function. For files of size 10KB, 100KB, and 1000KB, the results were as follows:

Operating System	**Delay 10KB (seconds)**	**Delay 100KB (seconds)**	**Delay 1000KB (seconds)**
OpenVMS	3.2	14.3	128.4
Tru64 UNIX	0.1	0.7	6.9

Only the third benchmark shows significant differences between OpenVMS and Tru64 UNIX. In reality, the poor performance of the buffered I/O of OpenVMS compared with Tru64 UNIX, can be explained by the slowness of the file system (RMS + ODS-5) and not by a bad implementation of the `BufferedInputStream()` function. In fact, the same order of difference is seen with the C program `unzip`.

In the case of the launch of Jetty under OpenVMS, the `Javadoc` deployment requires the most file system intensive use and is unsurprisingly the most costly. Of the 105 seconds taken for the launch, 96 seconds were used for decompressing the `javadoc.war` archive into 42 folders and 549 files.

Solutions exist to compensate for the file system slowness of OpenVMS, although no single solution alone suits all Java applications. In the case of Jetty, by default this application server deploys web applications (war archives) into a temporary folder where it also stores the bytecode of the servlets that are created in real time. The redirection of this temporary folder toward a RAMdisk decreases the launch time from 105 to 18 seconds and noticeably improves the response times of the web applications. Another solution that allows noticeable gains is to use only web applications that are deployed manually and with precompiled JPSs.

To summarize, Java is not poorly adapted to the OpenVMS environment. In reality, file system intensive applications are always slower on OpenVMS than on UNIX or Windows® systems, regardless of whether the applications were developed in Java. The only criticism that we can make of Java is that it frequently creates large numbers of application files (a bytecode per Java class).

Myth 3: Java is totally portable.

The creators of Java conceived an environment (VM language and standard libraries) that allows development of programs with maximum portability. In theory, it is the VM, not the developer, that deals with all the OS and hardware-related specificities.

Can we say, then, that Java is totally portable, as Sun implies with its slogan, “Compile once, run everywhere”?

The numerous tests and portings that we have performed in the OpenVMS environment show that Java programs are much more portable than those written in a compiled language. However, we are still a long way from verifying total portability. The reason is simple. The Java library creators, VM developers, and application developers are mainly from the UNIX world and, therefore, are unfamiliar with OpenVMS. (The OpenVMS VMs, Fast VM for Alpha, and HotSpot VM for Itanium systems, have been ported from Tru64 UNIX and HP-UX, respectively; the majority of Java applications were developed on and for UNIX platforms.)

Portability problems can be divided into two categories:

- Problems for the OpenVMS system engineer
- Problems for the Java application developer

The problems the system engineer encounters most often concern files and DCL, and these problems can be resolved without recompiling the application. For a description, see the *User Guide for Java for OpenVMS.*

Java deals only with files whose format is Stream-LF. Consequently, the system engineer must ensure that all the Java application files (bytecode, `jar` archive, data files, and so on) respect this format. Having originated in UNIX, the majority of Java applications use file names that do not respect ODS-2 file semantics. If this is the case then it is vital that the file system be migrated to ODS-5 format immediately, and that the Java application process is run with the correct parameters. For example, the following commands are often used:

```
$ SET PROCESS/PARSE=EXTENDED
$ DEFINE DECC$EFS_CASE_PRESERVE ENABLE
$ DEFINE DECC$ARGV_PARSE_STYLE ENABLE
$ DEFINE DECC$EFS_CHARSET ENABLE
$ DEFINE DECC$EFS_CASE_SPECIAL ENABLE
$ DEFINE DECC$ENABLE_GETENV_CACHE ENABLE
$ DEFINE DECC$POSIX-SEEK_STREAM_FILE ENABLE
$ DEFINE DECC$FILE_SHARING ENABLE
$ DEFINE JAVA$FILENAME_CONTROLS 8
```

In the preceding list, the first two commands force the process to respect case, and the last command defines the UNIX and OpenVMS file-name mapping rules that the VM must use. Sometimes choosing the good value for these logical names can be a delicate matter.

Java ignores the notion of file versions. Because of this, a good practice is to limit file versions to one on all files in the Java application file hierarchy. To do this, use the following commands on the root folder of the hierarchy:

```
$ SET DIRECTORY/VERSION LIMIT=1 disk:[rootfolder...]
$ SET FILE/VERSION_LIMIT=1 disk:[rootfolder...]*.*;*
```

The remaining task is to translate the application startup scripts into DCL procedures. You must use the foreign command `JAVA` to launch Java applications. This command can take either of the following two forms:

```
$ JAVA bytecode
$ JAVA -jar archive-jar
```

The bytecode name corresponds to the Java class name in the source code. Because Java is case sensitive (unlike DCL) the bytecode name must be quoted to deal with this difference. For example:

```
$ JAVA "MyClass"
```

The `JAVA` command accepts several optional parameters, including the `classpath`, which indicates the location of the bytecodes. These parameters can sometimes cause the final command line to exceed the DCL limit for commands within a procedure. For OpenVMS Version 7.3-2, this limit is set to 8192 bytes, and the `JAVA` command must be modified to respect this limit. OpenVMS provides various workarounds to accomplish this (for example, `option` file, `classpath` file; logical names such as `JAVA$ENABLE_ENVIRONMENT_EXPANSION`, `JAVA$CLASSPATH`, and so on).

The problems the Java developer encounters most often concern process management (such as fork, environment variables) and can be only resolved by modifying the source code and recompiling the application. These problems are illustrated by the iReport application, a reporting tool drawn from Microsoft® Access and Crystal Report.

The iReport tool uses an external program (for example, XPDF or Mozilla) to display the report in PDF or HTML format. This program runs within a detached process. On OpenVMS, the display works fine locally but fails remotely. The reason is that the X11 parameters (`display node`, `display transport`) are logical names that are not passed to the child process (display) by the parent process (iReport). We had to modify the source file (`iReportCompiler.java`) in order for the iReport process to pass the X11 parameters to the process controlling the display.

Use of the logical name `JAVA$EXEC_TRACE` can be useful for diagnosing portability problems related to process management. Once set to "true" in the LNM$JOB table, this logical name forces the VM to trace calls to the POSIX function `execv()` as well as its list of arguments.

Acknowledgments

The authors thank Tim Oakley for his French-to-English translation of this paper.

Trademarks

For more information

Useful links:

The Computer Language Shootout Benchmarks website: http://shootout.alioth.debian.org/
Documentation of Java on OpenVMS: http://h18012.www1.hp.com/java/documentation/
Jetty and iReport for OpenVMS: http://vmsfree.free.fr/freen/index.php

Contacts:
jeanyves.bourles@free.fr
t.uso@laposte.net