Appendix G
JIT Compilation versus AOT Compilation

Recall from Chapter 1 that the Java Virtual Machine provides a Just-In-Time (JIT) Compiler, which translates Java bytecode into native machine code at runtime. Plus, the JIT compiler profiles the code as it is running to discover methods (hot spots) where additional optimizations can be performed. Performance improvements can be significant for methods that are executed repeatedly. Note that Java's JIT compiler is part of the Java Virtual Machine, JVM, not the actual Java compiler that translates Java source files into bytecode. The term JIT typically applies to a virtual machine.

But many compilers use a different compilation model called Ahead-of-Time (AOT) compilation, whereby the source code is translated completely into a native executable format for a specific environment; e.g., the compiler (possibly together with a linker) creates a ".exe" file for a Windows/x86-64 computer. All translation and optimization are done at compile time, and the native code is run directly, not as part of a virtual machine. Practically all C and C++ compilers use AOT compilation.

So which compilation model is better, JIT or AOC? The answer is … it depends. Native executables created by AOT compilers typically start faster and run faster initially, but JIT compilers can often provide better overall optimization based on runtime execution patterns. Many server-based applications (e.g., web applications) are designed to run for extended periods of time – months to possibly years. For these types of applications, JIT compilation is an excellent choice. But for many applications, startup time and initial performance are more important, and for these applications, AOT compilation is usually a better choice. Compilers tend to fall in the second category of applications.

The three major applications discussed in this book are the CPRL compiler, the CVM (virtual machine), and the assembler for CVM assembly language. Complete source code for the latter two applications is available on the book's GitHub repository. Throughout development of the CPRL compiler, we assume that all three applications will run on the JVM. But none of these applications are designed to run for extended periods of time, and so it seems likely that they could benefit from AOT. Let's find out.

There are several tools that can be used to convert JVM applications into native executables, including the Java Development Kit (JDK) packaging tool `jpackage` and the GraalVM tool `native-image`. We will use the latter to create native applications for the compiler, the assembler, and the CVM. The discussion that follows will create native applications for Windows/x86-64 computers, but it is relatively straightforward to create native applications for Apple or Linux computers.

We start by assuming a Windows/x86-64 environment and a project setup as described in Appendix A. We also assume that GraalVM is installed and that its `bin` subdirectory is on the execution search path (included in the `PATH` environment variable). Then we create a subdirectory of the compiler project's `bin` directory (the one containing the scripts for compiling, assembling, and testing your CPRL test examples) named "`native`".

Using a text editor, add a script file named "make_cprlc.cmd" to directory native with the following contents.

```
@echo off

rem set config environment variables locally
setlocal
call ..\cprl_config.cmd

native-image -cp "%COMPILER_PROJECT_PATH%" -march=compatibility -O3 ^
edu.citadel.cprl.CompilerKt cprlc

rem restore settings
endlocal
```

The two key lines in the above listing are highlighted in bold. The "-march" and "-O3" options allow all features of the x86-64 with all optimizations for best performance. Note that the caret (^) character is the line continuation character for Windows command files. Typically you would just put everything on one line, but a single line was too long for the font and margins of this book.

For a bash/x86-64 environment, the script file "make_cprlc" (no ".cmd" suffix) would be written as follows, with the backslash (\) character serving as the line continuation character for bash scripts.

```
#!/bin/bash

# set config environment variables
source ../cprl_config

native-image -cp "$COMPILER_PROJECT_PATH" -march=compatibility -O3 \
edu.citadel.cprl.Compiler cprlc
```

In Windows, running make_cprlc.cmd from a command prompt will create the native executable file cprlc.exe. Follow a similar procedure to create native executables assemble.exe and cprl.exe.

Copy setPath.cmd and all test... files from the project's bin directory to the newly created native subdirectory. At this point the directory structure should look like the one shown on the next page (some files not shown).

Note that script files assemble.cmd, cprl.cmd, and cprlc.cmd in the parent directory have been replaced by native executables assemble.exe, cprl.exe, and cprlc.exe in the subdirectory. But since the other script files just call the executables by their base name (i.e., without the ".cmd" suffix), they will still run correctly from either directory. For example, when script file testCorrect_all.cmd calls cprlc, it will call either cprlc.cmd or cprlc.exe as appropriate.

Structure of the project `bin` directory for native executables is as follows:

```
bin                (the project bin directory)
 - assemble.cmd
 - cprl.cmd
 - cprlc.cmd
 - cprl_config.cmd
 - setPath.cmd
 - testCorrect.cmd
 – testCorrect_all.cmd
 - testEverything.cmd
 - testIncorrect_all.cmd
 - native          (subdirectory of bin)
    - assemble.exe
    - cprl.exe
    - cprlc.exe
    - make_assemble.cmd
    - make_cprl.cmd
    - make_cprlc.cmd
    - setPath.cmd
    - testCorrect.cmd
    – testCorrect_all.cmd
    - testEverything.cmd
    - testIncorrect_all.cmd
```

Now let's see if the native executable files offer improved performance. Open two command prompts. In the first command prompt, change directory to the project's `bin` directory and run `setPath.cmd` in that directory to ensure that it is included in the execution search path. Then run `testEverything.cmd`, observing how long it takes to run. If the compiler project is correct and complete, then all tests should run with the expected results.

In the second command prompt, change directory to the `native` subdirectory and run `setPath.cmd` in that directory to ensure that it is included in the execution search path. Then run `testEverything.cmd`, observing how long it takes to run. You should observe that `testEverything.com` ran much faster in the `native` subdirectory than it did in the project's `bin` directory, but let's measure.

Bash (and therefore Linux and Apple computers) has a `time` command that can be used to measure how long a given script takes to run. Windows PowerShell has a "cmdlet" (PowerShell term) called `Measure-Command` that does something similar, but the standard Windows Command Prompt doesn't seem to have a standard equivalent. There is an open-source utility called `ptime` that can be used for this purpose, but the web page for `ptime` says that it was last tested under Windows XP in 2005 and may be obsolete. Let's write our own utility for this purpose in Kotlin.

Omitting error checking for arguments, method `main()` for our timing program can be implemented similar to the following.

```
fun main(args: Array<String>)
  {
    println()
    println("===  " + args[0] + " ===")

    // for windows
    val pb = ProcessBuilder("cmd.exe", "/c", args[0])

    // for bash
    //val pb = ProcessBuilder("bash", "-c", args[0])

    pb.redirectErrorStream(true)

    val nanoseconds = measureNanoTime {
        val p = pb.start()
        val reader =
                BufferedReader(InputStreamReader(p.getInputStream()))
        var line: String?
        while ((reader.readLine().also { line = it }) != null)
            println(line)
      }

    ...   // convert nanoseconds to appropriate time unit
    ...   // write out elapsed time
  }
```

Using the approach outlined in this appendix, we can create a native executable named `wtime.exe` for our timing program.

The execution of the script file `testEverything.cmd` was timed using all three timing tools described previously in this appendix.

1. Windows PowerShell cmdlet `Measure-Command`
2. `ptime` (Yes, it still works in Windows 11!)
3. `wtime` (as described above)

All three gave very similar results, averaging more than 13 seconds for the non-native version versus less than 4 seconds for the native version.

For Windows PowerShell users, run these two commands separately in a PowerShell prompt.

```
$env:Path = "$PWD;" + $env:Path  # set path to include current dir
Measure-Command { testEverything | Out-Default }
```

So for Windows, the native versions of the compiler, assembler, and virtual machine proved to be much faster than the JVM versions when run against the test examples. But what about Linux and other Unix-based operating systems such as macOS? Using a process similar to the one described above, native applications were also created for Linux.

When script `testEverything` was run with these versions of the applications, the results were, disappointingly, slightly slower. Additional investigations showed that native executables for the compiler and assembler were faster, but the culprit was the native executable for the CVM, the virtual machine. For some reason, the native executable for the CVM generated by GraalVM for Linux ran more slowly than the JVM version.

As suggested by exercise 14 in Appendix B, the CVM was implemented for both Windows and Linux in the C programming language, and native executables were created for both Windows and Linux. The C version for Linux was compiled using the GNU `gcc` compiler, and the C version for Windows was compiled using the Microsoft C compiler in Visual Studio. The C versions proved to be the fastest in both Windows and Linux environments. The script `testEverything` ran about twice as fast in Linux with the compiled C version of CVM than it did with the GraalVM-generated native version. When tested on Windows, the compiled C version ran only a fraction of a second faster than the GraalVM-generated native version.

What about application size? When GraalVM generates a native executable for an application, it needs to include a lot of the JVM environment as well as the application-specific code. But, surprisingly, the GraalVM-generated version of the CVM was approximately twice as large as the GraalVM-generated versions of the compiler and the assembler, even though the source code for the CVM was smaller than the source code for the other two applications. One would expect that the C versions of the executables for the CVM would be much smaller. That was, indeed, the case for both the Windows and Linux versions as shown in the following table.

Sizes of Native Executables

	GraalVM Version	C Version
Windows	17,140 KB	197 KB
Linux	17, 774 KB	33 KB