Pedal to the metal: High-performance Java with GraalVM Native Image

How Native Image’s G1 GC garbage collector and profile-guided optimizations can help build fast, efficient, easy-to-distribute binaries for Java applications

by Oleg Šelajev

May 14, 2021

GraalVM Native Image can be a compelling platform for your Java cloud applications. As I wrote in “GraalVM: Native images in containers,” Native Image precompiles your applications ahead of time (AOT). This obviously removes the need to compile at the start of runtime, so you get apps that start almost instantly and have a lower memory footprint. This saves resources for the just-in-time (JIT) compiler infrastructure, class metadata, and so on.

Beyond the fast startup, there are different reasons why developers use native images with their applications. There’s the cloud friendliness of such deployments and obfuscating the code to improve security.

Figure 1 is an image often brought up when talking about performance and the different ways GraalVM can run your Java applications. You can see there are many axes, labeled with things people mean when saying “better performance.”
Sometimes better performance is about throughput and how many clients one instance of a service can handle; sometimes it's about serving individual responses as fast as possible, memory usage, startup, or even about comparing the size of the deployment because in certain scenarios, that can influence, for example, cold-start performance.

What's important is that with a few relatively simple tricks, plus using advanced GraalVM Native Image features, you can leverage all of these advantages for your applications.

In this article, I'll show you how to make the most of GraalVM Native Image technology for your applications.

**Creating a sample application**

Imagine that you have a simple sample application, which is a Micronaut microservice that responds to HTTP queries and computes prime numbers. It's a simple one-controller application that simulates the business logic by generating garbage collector pressure with some temporary objects by conveniently using the Java Stream API and the CPU to compute sequences of prime numbers very inefficiently: by trying all numbers as factors, including even numbers larger than 2.

Here's how you can create this app if you have the Micronaut command-line utility installed.

```
mn create-app org.shelajev.primes
cd primes
cat <<<'EOF' > src/main/java/org/shelajev/PrimeController.java
package org.shelajev;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import java.util.stream.*;
import java.util.*;
@Controller("/primes")
public class PrimesController {
    private Random r = new Random();
    @Get("/random/{upperbound}")
    public List<Long> random(int upperbound)
```
Now you have the sample app. You can run it or immediately build it into a native executable.

```
./gradlew build
./gradlew nativeImage
```

And then, you can run the app.

```
java -jar build/libs/primes-0.1-all.jar
./build/native-image/application
```

To test the application, you can open the page manually or run a `curl` command as follows, which will return to you a sequence of prime numbers less than 100:

```
curl http://localhost:8080/primes/random/100
```

However, to help illustrate the later stages of this article, you should download and install `hey`, a convenient HTTP load generator you'll use to assess peak performance.

Download the binary and put it on the `$PATH` (if you're not on Linux, grab the appropriate binary):

```
wget https://hey-release.s3.us-east-2.amazonaws.com/hey_linux_amd64
chmod u+x hey_linux_amd64
sudo mv hey_linux_amd64 /usr/local/bin/hey
hey --version
```

You can verify that it works and become acquainted with the output it provides by running the following command:
The output is longer than is reasonable to include here, but it prints a latency distribution diagram and a summary, such as the following:

```
  Summary:
  Total:   15.0021 secs
  Slowest: 0.1064 secs
  Fastest: 0.0001 secs
  Average: 0.0015 secs
  Requests/sec: 33703.8539
  Total data:  20062978 bytes
  Size/request: 20 bytes
```

The most important part you'll use for measurements is the `Requests/sec: 33703.8539` line, which shows the throughput of the application.

You'll do the measurements with limiting the heap size of the application to 512 MB, rather than allowing it to grow indefinitely. By default, Native Image sets the `-Xmx` option to 80% of the available memory to limit the heap size, which on a powerful test virtual machine would be overkill for this test application.

**Better memory management**

Since I'm talking about memory, a reduced memory footprint at runtime is one important metric where Native Image offers an improvement for running your application using a generic JDK.

The savings are mostly a one-time advantage because executables built with Native Image contain all the code in the application already compiled and all the classes analyzed. This allows you to leave out the class metadata and JIT compiler infrastructure.

However, the data set your application operates on takes a similar amount of memory, because the object layouts are similar on the JVM and in the native image. So if an application holds a few gigabytes of data in memory, Native Image will take a similar amount minus the 200 MB to 300 MB slice I talked about above.

Native Image obviously includes a runtime to support the application, which operates under the assumption that memory is managed and garbage is collected when needed. The implementation of the runtime used in native images, including the garbage collection, is from the GraalVM project.

The services mentioned above are written in Java, and since during the build of your application classes, dependencies and
JDK class library classes must be compiled anyway, the runtime is compiled together with your application. (The Serial garbage collector is a straightforward serial scavenger, which is optimized for throughput—not for minimizing latency.)

The garbage collector exposes the same memory configuration options for specifying heap sizes as the JDK exposes, for example: * -Xmx * for maximum heap size and * -Xmn * for young generation size. The -XX:+PrintGC and -XX:+VerboseGC options are also available if you feel the need to look behind the curtain or fine-tune the garbage collector for your particular workload.

If configuring the generation size is not your first preference, you could build the native image with the multithreaded G1 GC garbage collector. G1 GC is a performance-oriented feature included in GraalVM Enterprise, and it has a very straightforward configuration.

To enable G1 GC, pass the --gc=G1 parameter to the Native Image build process. Since you’re working with a Micronaut application and relying on its Gradle plugin for configuring and running the Native Image builder, specify the option in the build.gradle file.

Add the nativeImage configuration with the args line, as follows:

```groovy
nativeImage {
    args("--gc=G1")
}
```

And build the native image again.

```
./gradlew nativeImage
```

Call this version app-ee-g1 to have an easier time distinguishing the results. Before you run the tests, I’ll show you some other useful options to make the performance of a native image better.

**Better overall throughput**

There are several factors that affect the throughput of your applications. Some of the main ones, of course, are the nature of the workload, code quality, the quantity and characteristics of the data your code is crunching, latency of input and output, and so on. However, a better runtime or a better compiler can significantly speed up execution.

GraalVM Enterprise comes with a more powerful compiler—and it can create a profile of execution, similar to what a JIT compiler
does during the runtime of the application. The compiler can use
this profile for producing what’s called profile-guided optimization
(PGO) during AOT compilation. PGO can bring the throughput of
the resulting executable much closer to warmed up JIT numbers.

One thing to note here is that a JIT compiler’s best feature is
that it runs at runtime, which makes the data available to it
always relevant to the current workload. GraalVM AOT
compilation using a profile works best if the profile is collected
while running workloads similar to what you have in production.
This is usually easy to achieve with a well-designed
microservice. Here's how to do that for the sample application.

First, build an instrumented binary to use to gather the profile for
the PGO. The option enabling that is --pgo-instrument. You
can add it to the build.gradle configuration and build the
image normally with ./gradlew nativeImage, as follows:

```groovy
nativeImage {
    args("--gc=G1")
    args("--pgo-instrument")
}
```

Now you can build the application and run the same load-
generation tool.

```
./build/native-image/application
```

And then run the following command:

```
hey -z 15s http://localhost:8080/primes/rando
```

When it is stopped, the application will create the
default.iprof file in the current directory (unless it is
configured to do otherwise).

Now you can build the final image using the --pgo option, while
providing the correct path. Note that the path is going two
directories up: Micronaut builds the native image in the
build/native-image directory, and you’ll execute the
instrumented binary from the project’s home directory, as
follows:

```groovy
nativeImage {
    args("--gc=G1")
    args("--pgo=../../default.iprof")
}
```

After the build is complete and it stores the binary with the
descriptive app-ee-pgo name, you are ready to observe the
results.

The size of the binaries

Before you get to perhaps the most interesting comparison of the performance data, look at the executable sizes. The binaries are large. You can make them smaller.

Here are the sizes of the binary files from this simple app without any optimizations for the size.

```
$ ls -lah app*
-rwxrwxr-x. 1 opc opc  58M May  6 20:41 app-c
-rwxrwxr-x. 1 opc opc  73M May  6 21:14 app-e
-rwxrwxr-x. 1 opc opc  99M May  6 21:25 app-e
-rwxrwxr-x. 1 opc opc  80M May  6 21:47 app-e
```

I didn’t list the binary that’s instrumented for gathering the PGO profile, because it’s not really intended to be distributed and used in production, but for the sake of completeness, it’s around 250 MB.

The binary consists of two main parts: the precompiled code of the application and the data created during initialization of the classes at build time, which is called the image heap. Both are equally important to understand for utilizing native images effectively.

**The code part.** This part is easiest to grasp. This part contains all the classes and methods that needed to be included in the image because the static analysis found a possible code path to them or their inclusion was preconfigured with the explicit configuration.

The code part includes your classes, their dependencies, the dependencies’ dependencies, and so on up to the JDK class library classes and classes generated at build time. In other words, it’s all the Java bytecode that will be executed in the resulting executable.

Note that the code part does not include the infrastructure to deal with the bytecode loading, verification, interpretation, or JIT compilation. So, naturally, if something isn’t compiled ahead of time, it will not be available and executable at runtime.

**The image heap.** This part is a more unfamiliar concept for many developers. Think of a native image build process as running your application for a bit, initializing some necessary classes and their data, and saving the state of the application for future use. Then, when you run the executable in test or production, the initialized state is already prepared to be used and the application startup is instant. This state obviously needs to be written somewhere. That’s what is stored in the image heap.
You can observe how much space the classes and packages in your app contribute to the final executable size by using the reporting options (`-H:+DashboardAll`) during the native image build. Armed with this information, you can restructure the application to eliminate code paths that might make the image heap larger than necessary.

You also can trade a bit of startup speed for better packing, such as by using UPX, which stands for Ultimate Packer for eXecutables. UPX can compress the binary, and this works surprisingly well—often producing binaries that are around 30% of the size of the original. When the GraalVM team looked into using UPX, we found that a moderately high packing level around 7 is a good default.

The following is a sample result of applying UPX to one of the sample binaries here:

```
upx -7 -k app-ee-pgo
Ultimate Packer for eX
Copyright (C) 1996
UPX 3.96        Markus Oberhumer, Laszlo Moln
File size         Ratio      Format
--------------------   ------   ----------
83267776 ->  24043764   28.88%   linux/amd6
Packed 1 file.
```

The binary size dropped from 80 MB to 23 MB, and the app is still very responsive.

```
$ ./app-ee-pgo
Micronaut (v2.5.1)
20:33:44.838 [main] INFO  i.m.context.env.Def
20:33:44.852 [main] INFO  io.micronaut.runtim
```

Those are some of the ways to make small native images of your applications. You can of course optimize further by analyzing the reports and changing code, respectively—but that type of optimization gets into the area of diminishing returns pretty fast.

**How far can Native Image take you?**

I’ve shown you a few different ways to optimize the performance of the executables produced by Native Image: from using the more sophisticated and adaptive G1 GC, so you don’t have to manually tweak memory settings, to enabling the profile-guided
optimizations, to packaging the executable more efficiently for a smaller disk or container footprint.

You should now run the sample load-generation script to see if these optimizations make any difference. For this article, I ran the following 15-second tests three times with a heap limited to 512 MB:

- The default native image and GC from GraalVM Enterprise: *app-ee*
- The same with G1 GC: *app-ee-g1*
- And then with PGO on top: *app-ee-pgo*

```
./app-ee -Xmx512m
Summary:
  Total:    15.0023 secs
  Slowest:  0.1304 secs
  Fastest:  0.0001 secs
  Average:  0.0010 secs
  Requests/sec: 49770.7845

./app-ee-g1 -Xmx512m
Summary:
  Total:    15.0029 secs
  Slowest:  0.1388 secs
  Fastest:  0.0001 secs
  Average:  0.0010 secs
  Requests/sec: 51690.8255

./app-ee-pgo -Xmx512m
Summary:
  Total:    15.0023 secs
  Slowest:  0.1193 secs
  Fastest:  0.0001 secs
  Average:  0.0007 secs
  Requests/sec: 73391.9314
```

As you can see, the differences between the initial out-of-the-box build and the ones using G1 GC and PGO are striking.

Just for fun, I ran the same load on the same application running on OpenJDK. I used GraalVM based on JDK 11, with the exact version like this.

```
java -version
java version "11.0.11" 2021-04-20 LTS
Java(TM) SE Runtime Environment GraalVM EE 21
Java HotSpot(TM) 64-Bit Server VM GraalVM EE
```

For this comparison, I picked an arbitrary JDK distribution that builds on the same version 11.0.11 from [SDKMAN!](https://sdkman.io). Here are the results.

```
java -Xmx512m -jar build/libs/primes-0.1-all.jar
Summary:
  Total:    15.0019 secs
```
In this test, the best-performing native image is 16% faster than OpenJDK. This, of course, is not a completely fair test for a runtime that relies on JIT compilation and needs to warm up. Then again, 15 seconds and a million requests served is quite a bit of time, especially on a powerful machine with a lot of CPU capacity, so the JIT compiler can work in parallel with the application code.

In any case, you can see that native image performance can be comparable to running your application with the JIT compiler—while achieving better startup performance and being more appropriate in constrained environments or for microservices.

**Conclusion**

This article looked at different ways to improve the performance of native images without any code changes. Using the adaptive G1 GC, applying profile-guided optimizations, and packing the executable with UPX produced a very efficient microservice that’s about 20 MB in size, starts up in 20 milliseconds, and outperforms OpenJDK on the first 1 million requests served.

GraalVM Native Image is a very exciting technology for Java workloads in a cloud environment. Hopefully this article introduced you to some of the ways to use Native Image more efficiently without changing your application code.

**Dig deeper**

- [GraalVM: Native images in containers](#)
- [Cloud-agnostic serverless Java with the Fn project and GraalVM](#)
- [Java frameworks for the cloud: Establishing the bounds for rapid startups](#)
- [Building microservices with Micronaut](#)

---

**Oleg Šelajev**

Oleg Šelajev (@shelajev) is a developer advocate at Oracle Labs, working on GraalVM—the high-performance embeddable polyglot virtual machine. He organizes the VirtualJUG, the online Java User Group, and a GDG chapter in Tartu, Estonia. In his spare time, he is pursuing a PhD in dynamic system updates and code evolution. He became a Java Champion in 2017.