An Efficient Native Function Interface for Java

Lev Bagryansky

Meta data

- Authors: Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, Hanspeter Mössenböck.
- Conference: Principles and Practice of Programming in Java, 2013
- 11 citations.
- 575 Downloads.
- 10 pages.

Matthias Grimmer
Johannes Kepler University, Austria

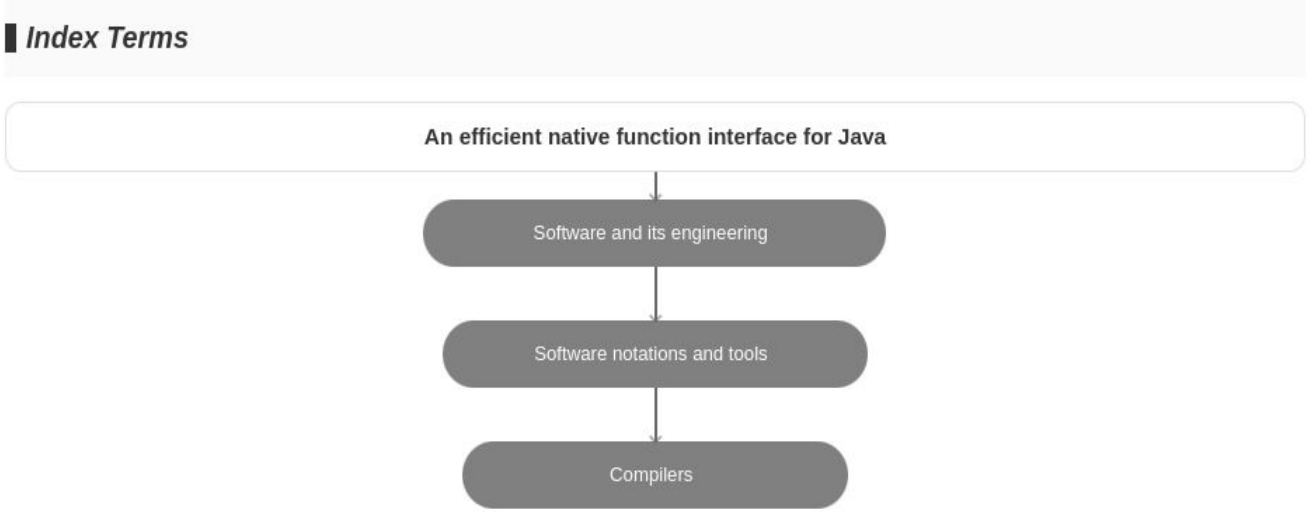
Manuel Rigger
Johannes Kepler University, Austria

Lukas Stadler
Johannes Kepler University, Austria

Roland Schatz
Oracle Labs

Hanspeter Mössenböck
Johannes Kepler University, Austria

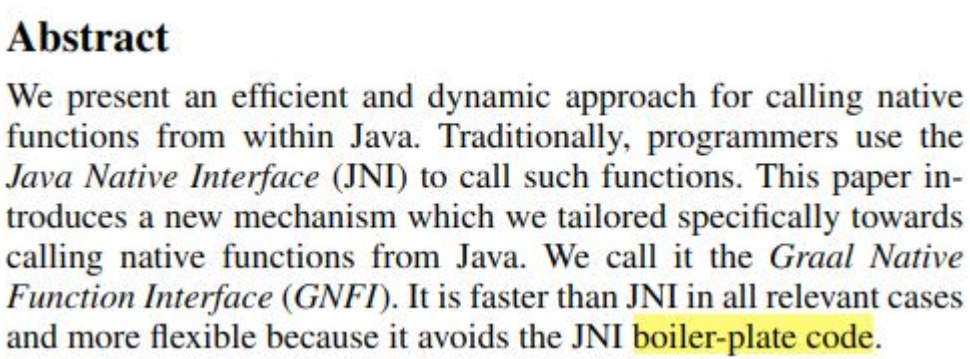
Paper layout



1. Abstract
2. Introduction
3. System Overview
4. The Graal Native Function Interface
5. The Call Stub
6. Native Function Calls in Interpreted Mode
7. Native Function Calls in Compiled Mode
8. Evaluation
9. Related Work
10. Conclusion
11. Acknowledgments (Not numerated)
12. References (Not numerated)

My personal assessment by criteria

- Problem statement. 😐
- Innovation. 👍
- Contribution. 👍
- Logical correctness. 👍
- Proof of statements. 👍
- Readability. 👍😐



Abstract

We present an efficient and dynamic approach for calling native functions from within Java. Traditionally, programmers use the *Java Native Interface* (JNI) to call such functions. This paper introduces a new mechanism which we tailored specifically towards calling native functions from Java. We call it the *Graal Native Function Interface* (*GNFI*). It is faster than JNI in all relevant cases and more flexible because it avoids the JNI boiler-plate code.

A brief overview of the content

While it is common to write JNI wrapper functions to call native target functions, this has several disadvantages:

- For invoking a native target function two calls have to be performed. The first call is from the Java application to the JNI wrapper and the second from the JNI wrapper to the native target function. In principle, a single call would suffice, namely from the Java application to the native target function.
- JNI has an additional overhead when setting up parameters. Especially when JNI accesses Java arrays, no access method guarantees not to copy the array [9, 11] before using it on the native side.
- The call to a native method is opaque to the JIT compiler, meaning that it cannot inline the call. We refer to this as a *compilation barrier*. Because of the compilation barrier, the JIT compiler cannot optimize the native code of the JNI wrapper function that sets up the parameters.
- The programmer has to implement, compile and link the JNI wrappers written in C/C++ instead of being able to call the native target function from Java directly.

```
double floor(double value);
```

Listing 1: Signature of an C function `floor`

```
NativeFunctionInterface ffi = Graal.
    getRequiredCapability(
    NativeFunctionInterface.class);
NativeLibraryHandle libraryHandle = ffi.
    getLibraryHandle("libMyMath.so");
NativeFunctionHandle functionHandle = ffi.
    getFunctionHandle(libraryHandle, "floor",
    double.class, double.class);
```

Listing 2: Obtaining a *GNFI* function handle

```
Object[] arg = new Object[1];
arg[0] =  new Double(1.5);
double result = Double.longBitsToDouble(
    functionHandle.call(arg));
```

Listing 3: Using the *GNFI* function handle to call a native target function

A brief overview of the content

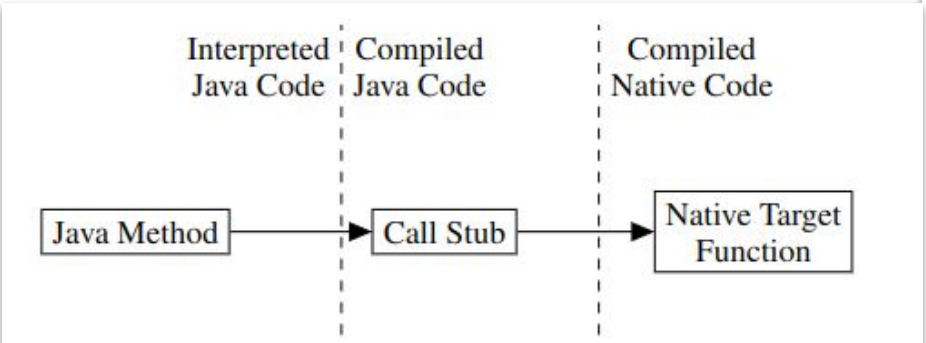


Figure 2: Layer gaps between the interpreted Java code and the native target function.

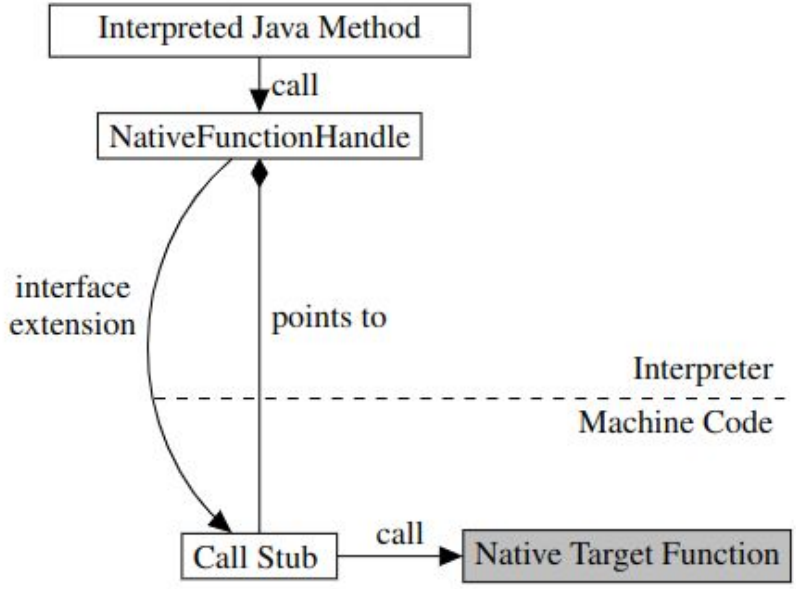


Figure 4: Execution in interpreter mode

Evaluation

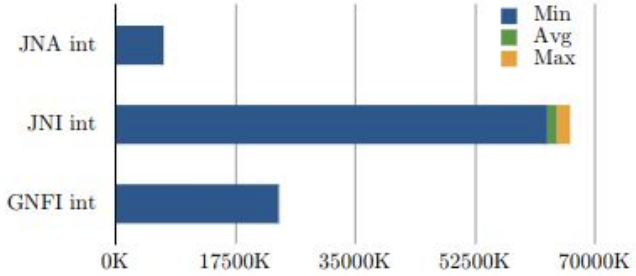


Figure 11: Performance of *nopFunction* (interpreted mode)

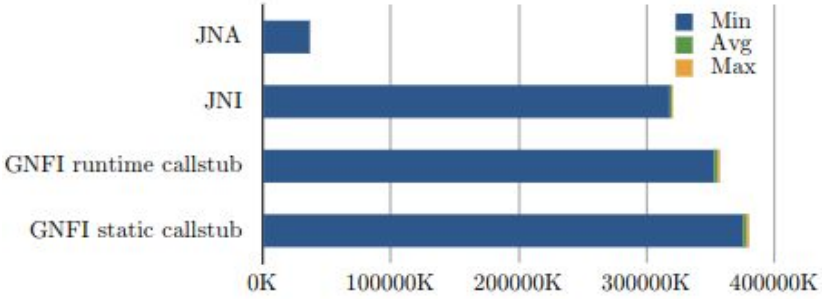


Figure 13: Performance of *nopFunction* (compiled mode)

Compared to JNI, *GNFI* has to go through more layers to perform the call. As we described in Section 5, *GNFI* has to bridge interpreted and compiled code, which impedes the performance. For the performance of an application, interpreted code plays a minor role. Hence, when we designed our implementation, we neglected the performance in interpreted mode and focused on performance in compiled mode.

For the micro benchmarks, *GNFI* with *static callstub* can inline the callstub (described in Section 6) and thus remove one call site. The JIT compiler can also remove the `args` array. This inlining explains the 6% speedup of the *static callstub* compared to the *runtime callstub*. *GNFI* with *runtime callstub* is faster than JNI because *GNFI* does a direct call to the call stub. Compared to JNI, *GNFI* does not set up any environment parameters, which explains the different performances.

Jblas Matrix Multiplication Benchmark

becomes negligible. The performance advantage of *GNFI* results from not having to copy the matrices.

While for $n = 10$ there was still a performance gap between compiled and interpreted Java code, this performance gap becomes almost non-existent for $n = 100$ and $n = 1000$.

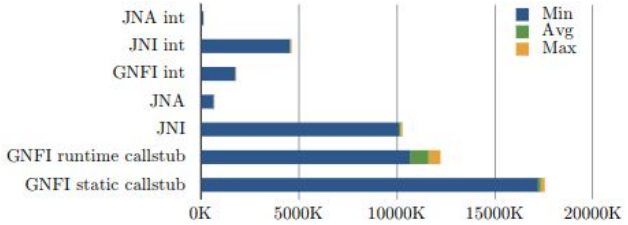


Figure 15: Performance of *jblas* for $n = 10$

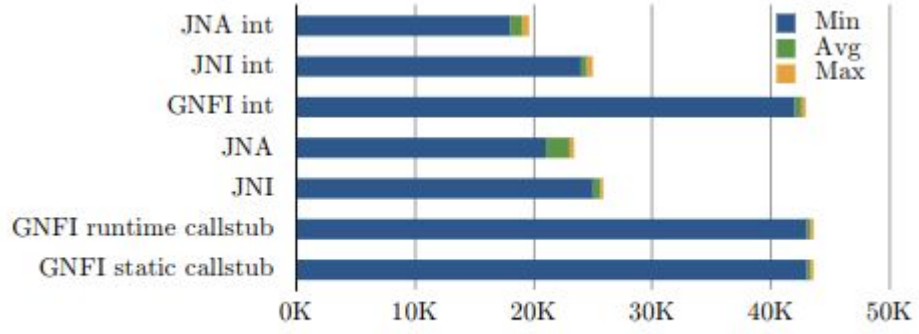


Figure 16: Performance of *jblas* for $n = 100$

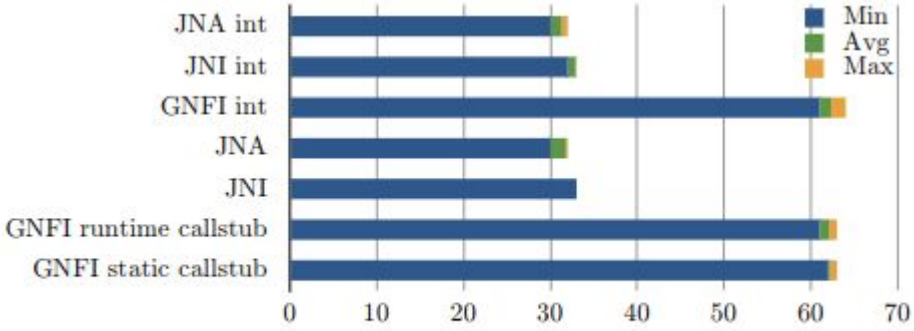


Figure 17: Performance of *jblas* for $n = 1000$

Paper’s disadvantages

- Some aspects are unclear(Why does jni copies array, why is jna so bad).
- I would like to see Related work at the beginning.
- What exactly JIT inlines.
- Limitations of new approach.
- Acceleration is not proved.

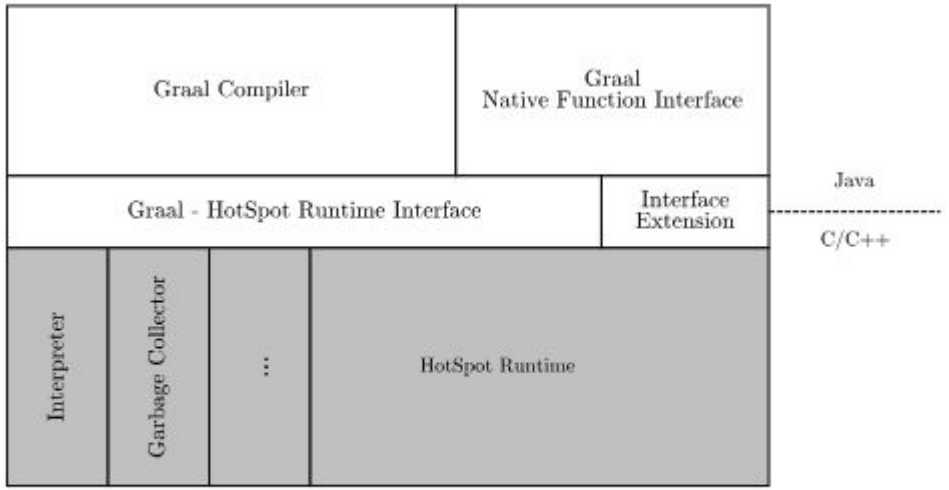


Figure 1: System architecture of the Graal VM.