Niklas Seppälä

Improving Java Performance With Native Libraries

Metropolia University of Applied Sciences

Bachelor of Engineering

Mobile Solutions

Bachelor’s Thesis

14 April 2024

Abstract

Author:	Niklas Seppälä
Title:	Improving Java Performance With Native Libraries
Number of Pages:	38 pages + 3 appendices
Date:	14 April 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Toni Spännäri, Senior Lecturer

The purpose of this thesis was to conduct in-depth research into available solutions for interfacing with native libraries from Java program, trying to identify the best solution for different situations. The study focused on JNI, JNA, and FFM API. The research was conducted for Profium.

The research relied heavily on Java language specifications and API documentations for source material, and supplementary benchmarks were created for comparing the different foreign function solutions in practice. Benchmarks evaluated Java wrapper-objects utilizing indexing library implemented in C programming language.

The outcome of this thesis was that while native libraries can be used to improve performance, it is not always required. The first step should be to find bottlenecks in existing Java code and to try optimizing it. Utilizing native functions is most suitable for larger computing tasks, which do not require communication back and forth between Java and native code.

Keywords: Java, performance, native, library, JNI, JNA, FFM

The originality of this thesis has been checked using Turnitin Originality Check service.

Tiivistelmä

Tekijä:	Niklas Seppälä
Otsikko:	Java-ohjelman Suorituskyvyn Parantamien Natiivikirjastoilla
Sivumäärä:	38 sivua + 3 liitettä
Aika:	14.4.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaajat:	Lehtori Toni Spännäri

Insinöörityön tarkoituksena oli tehdä laajamittainen tutkimus saatavilla olevista ratkaisuista natiivikirjastojen yhdistämiseen Java-ohjelmasta, ja yrittää tunnistaa paras ratkaisu erilaisiin käyttökohteisiin. Tutkimus keskittyi JNI, JNA FFM API ratkaisuihin. Insinöörityö tehtiin Profium Oy:lle.

Insinöörityössä hyödynnettiin vahvasti Java spesifikaatioita ja dokumentaatioita lähteinä, sekä ajettiin täydentäviä suorituskykytestejä eri ratkaisujen vertailuun. Suorituskykytestit testasivat Java-olioita, jotka kutsuivat C-ohjelmointikielellä kirjoitetun indeksointikirjaston funktioita.

Tämän insinöörityön lopputuloksena oli, että natiivikirjastojen hyödyntäminen Java-ohjelman suorituskyvyn parantamiseen on mahdollista, muttei aina tarpeellista. Ensisijaisen tärkeää on ensin etsiä huonon suorituskyvyn syynä olevat kohdat ja tehdä optimointeja Java koodissa. Natiivikirjastot sopivat parhaiten suurten data määrien prosessointiin ilman jatkuvaa edestakaista kommunikaatiota Java ja natiivikoodin välillä.

Avainsanat: Java, suorituskyky, natiivi, kirjasto, JNI, JNA, FFM

Contents

List of Abbreviations

1 Introduction 1

2 Java Native Interface 2

2.1 Native stack 3
2.2 Calling JNI code in Java Virtual Machine 4
2.3 Compiling JNI 5
2.4 Java types as JNI-types 6
2.4.1 Reference types 7
2.5 Error handling 8
2.5.1 Java Exceptions 8

3 Java Native Access library 10

3.1 Interface Mapping 12
3.2 Direct Mapping 13
3.2.1 Downsides 14
3.3 Library loading 14
3.4 Data type conversions 15
3.4.1 Unsigned integers 17
3.4.2 Structure 17

4 Foreign Functions and Memory API 19

4.1 Memory 19
4.1.1 Memory Segment 20
4.1.2 Arena 21
4.1.3 Data Alignment 23
4.1.4 Zero-length MemorySegments 24
4.1.5 MemoryLayout 25
4.2 Foreign Functions 28
4.2.1 Linker 29

5 Benchmarks 30

5.1 Native test library 30
5.2 Java implementations 31

5.3 Benchmarks 32
5.3.1 Cold start benchmark 33
5.3.2 Garbage collection pressure benchmark 33
5.3.3 Throughput benchmark 34
5.4 Analysis 35
6 Summary and conclusions 35
References 37
Appendices
Appendix 1: C Header file of the native word index library
Appendix 2: WordIndex.java
Appendix 3: JNI bindings embedded in wordindex C-library

List of Abbreviations

JVM:	Java Virtual Machine. Provides a language runtime for Java code to be executed on.
JIT:	Just-in-time. Type of compiler that compiles intermediate code to native instructions.
JDK:	Java Development Kit. Provides tools to develop and run Java programs.
JNI:	Java Native Interface. The default solution for interfacing with foreign functions. Included since JDK 1.1.
JNA:	Java Native Access. Third party library providing pure Java interface for interfacing with foreign functions. Focuses on ease of use over performance.
FFM API:	Foreign Function and Memory API. Pure Java replacement for JNI. Available since JDK 22.
ABI:	Application Binary Interface. Interface defining how code binaries should be loaded and executed.
JAR:	Java archive. Zip package containing Java class files and metadata.
GCC:	GNU Compiler Collection. Collection of compilers, including C compiler.
Java SE:	Java Platform, Standard Edition. Java platform that contains most used Java APIs
G1:	Garbage-First collector. Generational Java garbage collector.

1 Introduction

Rapid development in hardware performance, and reduction in costs have caused a shift in software development priorities for commercial products. Focus on performance in processing speed and memory efficiency has been largely overshadowed by development speed and memory safety. Java and its large ecosystem can usually be seen as a good fit for these requirements, however, some performance critical applications, such as databases, web servers, and search indexes often require more control than Java is willing to provide.

Native code executes instructions directly on the CPU and is compiled and optimized for specific runtime platform. This is machine code, native to the platform it was compiled to. Java has an additional layer between the code and the CPU, called Java Virtual Machine (JVM). JVM interprets the compiled Java bytecode to platform dependent instructions for the CPU to execute. Having a virtual machine and intermediate bytecode abstraction between code and hardware allows Java code to be executed in any environment where JVM is installed, but this additional layer comes with performance penalty.

The gap between native code and Java code performance has narrowed after the introduction of just-in-time (JIT) compilation. While JVM executes Java bytecode, it looks for frequently called methods that could benefit from being compiled directly to machine code. This removes the bytecode abstraction and improves Java performance, but the difference can still be enough to make a difference in performance critical applications.

Another major difference between native code and Java is the way memory is allocated. In Java, all structured data is allocated on garbage collected heap. The heap requires frequent maintenance work by the garbage collector, which will eventually pause all application threads while it frees and reorders the objects around the heap.

It is possible to delegate computationally heavy tasks outside of the Java code and the restrictions that come with the JVM, by calling native library functions. This requires crossing the bridge between JVM and native code, which is by no means a free operation.

The goal of this thesis is to conduct in-depth research into three available solutions for interfacing with the native libraries: Java Native Interface (JNI), Java Native Access (JNA), and Foreign Function and Memory API (FFM API). It attempts to provide the reader with a deep understanding of how the different solutions work, what they provide, and how to apply them, so better judgment can be used for determining which, if any, solutions would be the best for the reader’s application.

Theoretical research is supplemented by conducting microbenchmarks, focusing on processing speed and memory footprint. Benchmarks use a purpose-built native C-library and a Java project that contain implementations for all aforementioned solutions for interfacing with native libraries, as well as pure Java implementation to act as a baseline for the comparisons drawn from the results.

The subject for this thesis was requested by Profium, related to Profium Sense™ in-memory graph database development. Profium Sense™ provides low latency queries on large and complex data sets.

2 Java Native Interface

Java Native Interface (JNI) is Java’s version of foreign function interface, available in most distributed JVMs. Java Native Interface allows Java code running in JVM to interoperate with native libraries and applications, written in programming languages such as C or C++ [1].

JNI is defined as a public interface for JVM vendors. Being isolated from the rest of the JVM, it makes no assumptions on how the JVM is implemented. This

comes with an important benefit: programmers integrating a native library to Java can assume that the JNI code will run on any JVM, that supports the Java Native Interface [1].

JNI has lasted the test of time, first released as part of JDK 1.1 in 1997 [2]. This, combined with the fact that it is shipped with the Java Development Kit (JDK) as the default foreign function interface in Java, has solidified its popularity and widespread adoption over the past decades. JNI might be one of most stable parts in the Java ecosystem and has remained practically the same in new Java releases since its introduction.

Java Native Interface enables not only Java code to interact with native code, but also native code to interact with the JVM. JNI has wide range of features allowing native code [1] to:

- Create, read, update, and delete java objects.
- Call both static class methods and object methods.
- Catch and throw Java Exceptions
- Load and inspect classes.
- Perform runtime typechecking.
- Embed whole Java Virtual Machine in a native application.

2.1 Native stack

Java Virtual Machine Specification dictates [3, p. 14] that any JVM implementation supporting native JNI methods, must provide a separate native stack structure for it to be able to call native code. Unlike Java stack, which consists of stack frames that can be heap allocated, native stack must exist as a contiguous block of memory. This type of stack is often referred as “C stack” , and it must abide the platform-dependent ABI (Application Binary Interface) calling conventions for it to be able to interoperate with native code correctly. Differences between Java stack and native stack structures are illustrated in Figure 1.

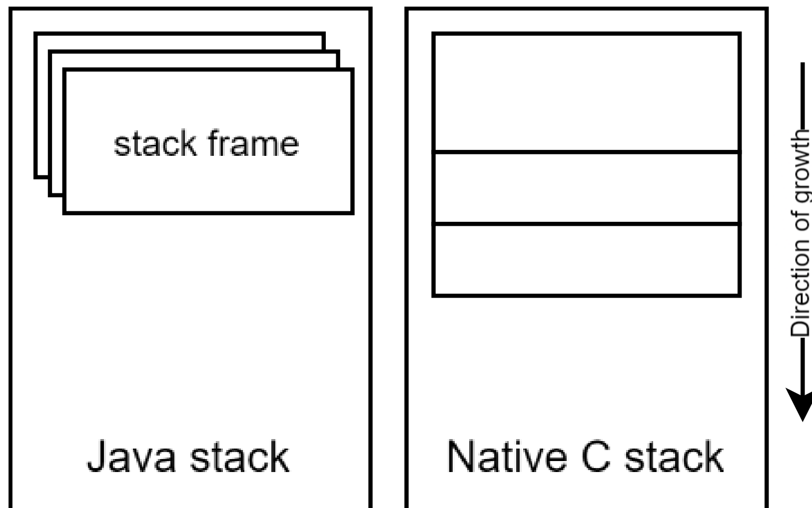

Figure 1. Java stack and native “C style” stack

JVM specification states [3, p. 14] that native stack can be either a fixed size, or it can expand as the executed computation requires. In the case where fixed size limit is reached, JVM will throw StackOverflowError, just like with Java code. If stack size is not limited to a fixed amount and available memory runs out, JVM will throw OutOfMemoryError.

2.2 Calling JNI code in Java Virtual Machine

When a thread calls a Java method, a new stack frame gets created, populated with method arguments, and pushed to the Java call stack by the JVM [4, p. 9], and the virtual machine executes the method. When calling Java method with native specifier, this flow differs significantly.

Java Virtual Machine Specification states [3, p. 526] that when a Java bytecode call instruction, for example invokevirtual 182 (0xb6), an instruction for calling instance methods, detects that method is native, it first checks if the native function providing the native implementation is already bound, and if not, it will bind it to the method. Native stack is constructed, and method arguments are popped from the current Java stack frame and get passed to the native method as direct arguments, instead of creating a new Java stack frame for the method. If the native method returns a value, it will get converted from native type to

corresponding Java type and pushed to the current Java stack as the method call result. If the library binary was not loaded before making the call, JVM will throw UnsatisfiedLinkError [5, p. 647].

2.3 Compiling JNI

Creating JNI bindings can be divided into two tasks. First, Java code needs to be compiled into class files, and header file extracted from native method declarations. Listing 1 shows NativeMalloc Java class, which defines static native method NativeMalloc.malloc(long).

```
// java class: SimpleMalloc.java
public class SimpleMalloc {
    public static native long malloc(long nbytes);
}


$ javac SimpleMalloc.java -h jni/include

// C header file: jni/include/jnidemo_SimpleMalloc.h
…
/*
 * Class:     jnidemo_SimpleMalloc
 * Method:    malloc
 * Signature: (J)J
 */
JNIEXPORT jlong JNICALL Java_jnidemo_SimpleMalloc_malloc
  (JNIEnv *, jclass, jlong);
…
```

Listing 1. Java class NativeMalloc declares native method malloc. Java compiler is used to compile the class and -h flag is used to extract C header file with matching JNI function declaration.

In this example, JNI bindings are embedded directly into the target library. If JNI bindings were needed for 3rd party library that could not me modified, the result would be separate binding library and target library.

The second task is to implement native bindings in C, C++, or Assembly, and compile it with languages compiler as shared or static library, as demonstrated in Listing 2. JDK header files must be available for compilation. The resulting binary must then be moved somewhere Java can find and load it at runtime. Native libraries can also be embedded in JAR files. Listing 3 shows an example of loading the created library and calling native method.

```
// C source: mymalloc.c

#include "my_simple_malloc.h"
#include "jni/include/jnidemo_SimpleMalloc.h"

JNIEXPORT jlong JNICALL
Java_jnidemo_SimpleMalloc_malloc(JNIEnv *env, jclass , jlong bytes) {
    return simple_malloc(jlong);
}


$ gcc -c -Wall -Wextra -Wpedantic -fPIC \
        -I"$JAVA_HOME/include $JAVA_HOME/include/linux" \
         mymalloc.c

$ gcc -shared -o mymalloc.so mymalloc.o
```

Listing 2. GCC C compiler is used to compile C source code to object file and create shared library mymalloc.so.

```
public static void main(String[] args) {
    System.load("mymalloc.so");
    final long ptr = SimpleMalloc.malloc(64L);
}
```

Listing 3. JNI compatible native library "mymalloc.so" is loaded, and native function is invoked.

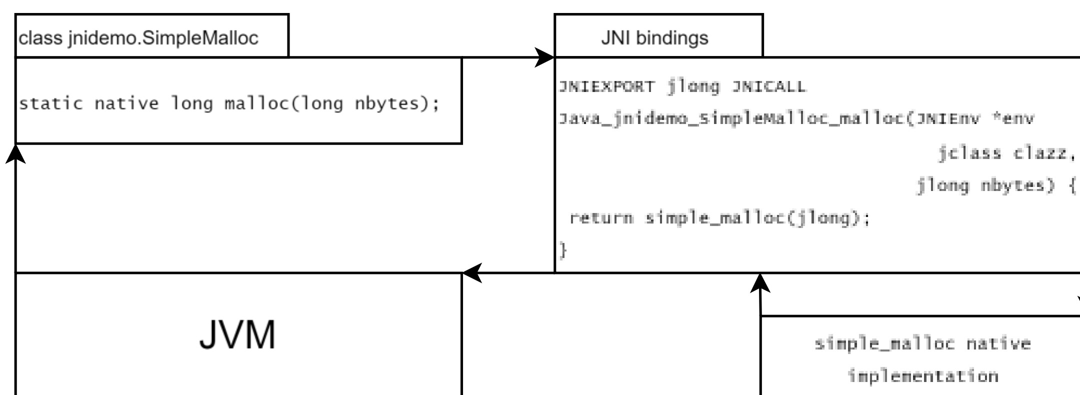

Figure 2. Java Virtual Machine calls static native method via JNI binding.

2.4 Java types as JNI-types

To guarantee that native JNI binding code uses the same byte width as in Java, JNI header file jni.h exposes type definitions, mapping platform dependent C types to Java primitives with matching attributes. Table 1. lists Java types as JNI types, with their native presentations. JNI types are aliases to platform dependent C-types [1].

Table 1. Java primitive types with native width, and corresponding JNI-type. JNI-types are C type definitions for common C primitive datatypes [1].

Java type	Native presentation	JNI-type	C-type
boolean	8-bits (unsigned)	jboolean	unsigned char
byte	8-bits (signed)	jbyte	char
char	16-bits (unsigned)	jchar	unsigned short
short	16-bits (signed)	jshort	short
int	32-bits (signed)	jint	int
long	64-bits (signed)	jlong	long long
float	32-bits	jfloat	float
double	64-bits	jdouble	double

2.4.1 Reference types

Java Language Specification [3, p. 64] defines reference types to contain class types, interface types, type variables and arrays, and that reference values are pointers to objects. It is also stated in the language specification [3, p. 68] that all Java classes are child classes of java.lang.Object class, meaning all reference types can be passed as java.lang.Object type. To achieve same type compatibility in JNI, it is stated in Java SE 22 Native Interface Specification [1] that all JNI-types are defined to be the same type as JNI-type jobject. Table 2. displays Java reference types to JNI-type definitions.

Table 2. Java reference types mapped to corresponding JNI-types. [1]

Java type	JNI-type
java.lang.Class	jclass
java.lang.String	jstring
java.lang.Object	jobject
java.lang.Throwable	jthrowable

Java type	JNI-type
java.lang.Object array	jobjectArray
primitive boolean array	jbooleanArray
primitive byte array	jbyteArray
primitive char array	jcharArray
primitive short array	jshortArray
primitive int array	jintArray
primitive long array	jlongArray
primitive float array	jfloatArray
primitive double array	jdoubleArray

2.5 Error handling

Java Native Interface Specification warns [1] that JNI interface functions have no sanitation for invalid pointer arguments. For example, de-referencing null pointer in JNI-code will cause JVM crash. It is argued in the specification, that by this design, JNI ensures as much performance as possible, and emphasizes diligent programming methods when calling JNI interface functions.

2.5.1 Java Exceptions

JNI native code can catch Exceptions raised by Java code. JNI code can also create and propagate Exceptions back to the JVM.

After an exception has been raised from Java code, most JNI functions cannot be safely called without clearing the existing exception. Exception can be cleared by calling JNI interfaces ExceptionClear() function [1].

To ensure the ability to reset the program state back to normal after an exception has occurred, JNI guarantees [1] that following functions are always safe to call:

- jthrowable ExceptionOccurred(JNIEnv*, jobject)
- void ExceptionDescribe(JNIEnv*)
- void ExceptionClear(JNIEnv*)
- jboolean ExceptionCheck(JNIEnv*)
- void ReleaseStringChars(JNIEnv*, jstring, const jchar*)
- void ReleaseStringUTFChars(JNIEnv*, jstring, const char*)
- void ReleaseStringCritical(JNIEnv*, jstring, const jchar*)
- void ReleaseIntArrayElements(JNIEnv*, jintArray, jint*, jint)
- void ReleasePrimitiveArrayCritical(JNIEnv*, jarray, void*, jint)
- void DeleteLocalRef(JNIEnv*, jobject)
- void DeleteWeakGlobalRef(JNIEnv *, jweak);
- jint MonitorExit(JNIEnv*, jobject)
- jint PushLocalFrame(JNIEnv*, jint)
- jobject PopLocalFrame(JNIEnv*, jobject)
- jint DetachCurrentThread(JavaVM *)

These functions are useful in debugging the exception cause, enabling recovery if exception was not considered fatal. Being able to do this without clearing the exception is useful in a case where exception handling is done in Java code instead of JNI code. Returning from native function without clearing the exception will cause JVM to throw it, and java try-catch method to handle it.

Java Native Interface Specification [1] states that even though JNI uses the shared monitor with Java code, when the monitor is acquired by JNI MonitorEnter() function, it can only be released by JNI MonitorExit() function. Imagine the case (listing 4) where JNI code has acquired a monitor before interacting with Java object in synchronized manner, in a multithreaded context. Then, an object method throws an exception, indicating that called procedure was not accepted by the logic in objects Java class. If the native code would just swallow the exception and return without releasing the monitor, the next attempt to acquire it would result in permanent deadlock, both from Java code and JNI code.

```
env->MonitorEnter(env, obj);

int result = calc_result(env, obj);

jthrowable exception = ExceptionOccurred(env);
if (exception) {
    env->MonitorExit(env, obj);
    // Propagate exception to JVM
    return;
}

// function continues with more steps
```

Listing 4. JNI native code that releases object monitor after Exception has been detected, and returns without clearing the exception, propagating it to the JVM.

3 Java Native Access library

Java Native Access (JNA) is a popular open-source alternative for JDK's default Java Native Interface. First released in 2007, it has been in active development by hundreds of contributors ever since. Distributed under LGPL 2.1 and from 4.0 release onwards Apache 2.0 license, JNA offers distributions for all JVM versions since Java SE 1.4 [6].

Some notable open-source projects using JNA [6] are:

- Apache Cassandra, a large-scale NoSQL database
- Intellij IDEA, an IDE
- Apache NetBeans, an IDE
- Elasticsearch, a large-scale distributed search engine

JNA-library offers its user a pure Java implementation for interfacing native code [6]. User defines a Java interface (listing 6) with desired native function declarations, matching corresponding declarations in the native library (listing 5). The interface must extend com.sun.jna.Library parent interface. A concrete instance of the Library interface is then generated at runtime by JNA-library using Java reflection when native library binary is loaded [7]. JNA uses libffi, a C library, to dynamically invoke the native functions of the loaded library, according to platform dependent calling conventions [7]. Before the library function can be called by libffi, JNA performs type conversions from Java datatypes to native datatypes [7].

```
// wordindex.h
WordIndex *file_word_index_open(const char *path,
                                enum index_analyzer analyzer,
                                size_t capacity, size_t word_buffer_size,
                                bool compact);

void file_word_index_close(WordIndex *index);
```

Listing 5. Native open and close function prototypes from C Header file of wordindex native library.

```
// JNAWordIndexLibrary.java
public interface JNAWordIndexLibrary extends com.sun.jna.Library {

    com.sun.jna.Pointer file_word_index_open(String path, int analyzer,
                                             long capacity, long buffSize,
                                             boolean compact);

    void file_word_index_close(com.sun.jna.Pointer handle);

    public abstract class Impl {
        private static volatile JNAWordIndexLibrary instance;

        public static void load(String libPath) {
            if (instance == null) {
                synchronized (JNAWordIndexLibrary.Impl.class) {
                    if (instance == null) {
                        instance = Native.load(libPath,
                                               JNAWordIndexLibrary.class);
                    }
                }
            }
        }

        public static JNAWordIndexLibrary get() {
            if (instance == null) {
                throw new IllegalStateException("Library is not loaded");
            }
            return instance;
        }

        public static JNAWordIndexLibrary get(String libPath) {
            load(libPath);
            return instance;
        }
    }
}
```

Listing 6. JNA native bindings for native open and close functions declared in wordindex.h (listing 1). Interface includes inner JNAWordIndexLibrary. Impl class using singleton pattern to store and access callable proxy-object generated by JNA-library.

```
public static void main(String[] args) {
    com.sun.jna.Pointer handle = JNAWordIndexLibrary.Impl
        .get("wordindex.so")
        .file_word_index_open(
            args[1], new EnglishAnalyzer().asNative(),
            10000, 8192, false
        );

    JNAWordIndexLibrary.Impl
        .get()
        .file_word_index_close(handle);
}
```

Listing 7. Example code for loading native library “wordindex.so” and calling native open and close functions.

The development goal for JNA-library is focusing on providing simple to use solution for calling foreign functions from Java code, and not focusing solely on performance. The overhead for calling native functions is usually an order of magnitude greater than custom made JNI implementations [8].

3.1 Interface Mapping

Interface Mapping is the default method JNA uses when calling native methods from Java code. JNA generates a proxy object using Java reflection that routes all calls to the defined library Java interface methods through invoke() method in Library.Handler class [7]. Handler looks for right proxy Function object, which represents the function in the native library. Once the Function object is found, it is called with arguments provided for the interface method call. Proxy object holds all metadata needed for type conversions before and after the actual native method call [7].

The native libffi library used by JNA requires description of native function arguments and return types. To construct platform-specific native stack suitable for the native function call, libffi will also do type conversions if necessary [7]. Once libffi completes the native call, it copies the return value into JNA provided native buffer, from where the JNA can convert the value back into Java object, or primitive value [7].

Interface Mapping provides convenience for the developers, but it suffers from all the overhead required before native call gets invoked [8]. Reflection is associated with poor performance in Java, and JNA makes use of reflection API for every native call. Also, the fact that JNA Structures are not backed by native memory, nor the Java Object, but a combination of the two, requires JNA to copy values between the two.

3.2 Direct Mapping

JNA library addresses the substantial performance cost of using interfaces and runtime Java reflection to invoke native code with an alternative called Direct Mapping. With this method, the performance gap to JNI is almost nonexistent [7]. In Direct Mapping, JNA creates code stubs for calling each registered native method only once, with complete type information of the function invocation [9], eliminating the runtime reflection performance penalty.

Calling directly mapped native method, JNA will attempt to pass Java stack frame to native function [7]. The more non-primitive arguments the method signature has, the more work needs to be done to convert them into native presentation before the stack can be passed [7]. To achieve the best performance, one should only use primitives or use Pointer arguments [7].

Bindings to native functions can be object methods or static class methods and must be decorated with Java native keyword. Native library is not loaded with Native.load(String) method, but its functions are registered with Native.register(String) [9]. Code snippet (listing 8) shows example of registering and calling math functions sin() and cos() from platforms native C-standard library.

```
import com.sun.jna.*;

public class HelloWorld {

    public static native double cos(double x);
    public static native double sin(double x);

    static {
        Native.register(Platform.C_LIBRARY_NAME);
    }

    public static void main(String[] args) {
        System.out.println("cos(0)=" + cos(0));
        System.out.println("sin(0)=" + sin(0));
    }
}
```

Listing 8. Example code for declaring and using directly mapped cos- and sin-functions available from C standard library [9].

3.2.1 Downsides

Direct Mapping is still missing some features of the more common Interface Mapping. Variadic arguments are not supported and instances of Java.nio.Buffer parent class cannot be used as return types in native method declarations. Also, Direct Mapping does not support [9] following array types for:

- com.sun.jna.Pointer
- com.sun.jna.Structure
- java.lang.String
- com.sun.jna.WString
- com.sun.jna.NativeMapped

3.3 Library loading

JNA-library has multiple different conventions to helping JNA to discover native library used at runtime. The preferred way is to set jna.library.path Java system property to point at the binary path on the local filesystem [10]. System property can be set as JVM runtime argument (listing 9), or at execution time in Java code (listing 10). Other options are to make sure binary location is defined in PATH, LD_LIBRARY_PATH or DYLD_LIBRARY_PATH, depending on platform, or including library in Java class-path [10].

When including binary in class-path, JNA uses {OS}-{ARCH}/{LIBRARY} pattern for finding it [10]. {OS}-{ARCH} section in the pattern defines the runtime platform:

- win32-x86 - Windows
- linux-amd64 - Linux
- darwin - macOS

If the library is within a jar file, it is automatically extracted when the library is loaded.

```
-Djna.library.path=”/usr/local/lib”
```

Listing 9. JVM argument that specifies path where JNA can discover native library binaries.

```
java.lang.System.setProperty(“jna.library.path”, ”/usr/local/lib”)
```

Listing 10. Java-code specifying where JNA can discover native library binaries, by setting a system property at runtime. System property must be set before JNA attempts to load the library.

3.4 Data type conversions

Unlike JNI, JNA provides some automated marshalling when calling native function, and unmarshalling for return value at runtime. For example, an instance of Java String class gets converted to C-string. Since Java characters are two bytes wide, the underlying character arrays elements get truncated to single byte width, and extra slot gets allocated for NULL-byte string termination. Table 3 contains datatype mappings for C primitive types to Java types used in JNA method signatures.

Table 3. JNA automatic C-type primitive type conversion to JNA Java types. [11].

C-type	Native presentation	Java type
char	8-bit integer	byte

C-type	Native presentation	Java type
short	16-bit integer	short
wchar_t	16/32-bit character	char
int	32-bit integer	int
int	boolean flag	boolean
enum	enumeration	int (usually)
long	platform-dependent 32/64-bit integer	NativeLong
long long	64-bit integer	long
float	32-bit floating point	float
double	64-bit floating point	double
const char*	C-string (32- or 64-bit pointer)	String
const wchar_t*	Wide C-String (Unicode)	WString
char**	C-String array	String[]
void*	platform-dependent 32- or 64-bit pointer	Pointer, Buffer
typed pointer (T*)	platform-dependent 32- or 64-bit pointer	Instance of PointerType parent class
void* (array)	platform-dependent 32- or 64-bit pointer	Pointer[]
struct	structure by value	Structure
struct*	32- or 64-bit pointer	Structure
struct[]	contiguous memory	Structure[]
union	union by value	Union
union*	platform-dependent 32- or 64-bit pointer	Union
union[]	contiguous memory	Union[]
void (*func)(void)	function pointer (Java or native)	Callback

3.4.1 Unsigned integers

Java is by default lacking primitive types for unsigned integers, and JNA library does not provide unsigned integer types by default [11]. One way is to do unsigned value conversion by hand. JNA library users can assign the corresponding integer’s two's-complement representation to the signed type of the same size. JNA also offers an abstract parent class IntegerType, where user can derive unsigned integer types. When implementing unsigned integer type, user should consider that the type of the field holding the unsinged value is always long. Listing 11 shows an example of unsigned 32-bit integer implemented by extending IntegerType class.

```
class U32 extends com.sun.jna.IntegerType {
    public U32() {
         super(4, true);
    }
}
```

Listing 11. Custom class U32 that extends the JNA IntegerType base class. Type U32 can be used to represent unsigned 4-byte integer.

3.4.2 Structure

For working with C-style structs, JNA offers a base-class for users to inherit from, com.sun.jna.Structure [11]. An instance of Structure is a Java object that models the corresponding C-style struct in Java code, but it also holds the native memory, accessed by native code. Before calling a native method, JNA allocates the memory, and writes the values from the object in correct order and alignment matching the native struct. When native method returns, JNA reads the native memory and updates the Java object [11]. Java class that inherits from Structure should be public, have a no-argument constructor [11] and fields related to native struct must be public [12].

By default, when Structure is used as a return, or parameter type, Structure gets passed to and from native method by reference, as a pointer to the Structure [12]. This prevents excessive copying of data to and from native methods. Default pass-by-reference policy can be overridden by tagging the

class inheriting Structure with Structure.ByValue or Structure.ByReference interfaces [12], demonstrated in listing 12.

```
typedef struct Vector2 {
    float x;
    float y;
} Vector2;

@Structure.FieldOrder({"x", "y"})
class Vector2 extends Structure {
    public static class ByReference extends Vector2
                                implements Structure.ByReference {}
    public static class ByValue extends Vector2
                                implements Structure.ByValue {}
    public float x;
    public float y;
}
```

Listing 12. Native struct Vector2 with x and y fields. JNA Structure Vector2 with x and y fields. Field order specified with FieldOrder Annotation. Static child classes Vector2.ByReference and Vector2.ByValue.

Structure’s native memory byte alignment derived from platform automatically. Alternatively, this behavior can be overridden by explicitly setting the alignment in the Structure’s constructor by calling Structure.setAlignType(int alignType). Available alignment types [12] are:

- Structure.ALIGN_DEFAULT – for platform dependent.
- Structure.ALIGN_GNU – for GNU Compiler Collection alignment.
- Structure.ALIGN_MVC – for Microsoft Visual Compiler alignment.
- Structure.ALIGN_NONE – for packed structs with no padding.
- Structure.CALCULATE_SIZE – Aligns to 8-byte boundary.

For JNA to know how to set object’s fields in native memory, field order must be specified. Declaring fields in Java class in the same order as native struct will usually work, but Java Virtual Machine Specification makes no remarks on which order JVM returns fields with reflection [12]. The recommended way is to use Structure.FieldOrder annotation (listing 12), but Structure.getFieldOrder method can also be overridden.

JNA Structure has special handling for volatile and final fields. When some outside actor, like native thread or hardware updates native memory outside of the JNA native call, the change will not be reflected to the Java object [12].

When JNA makes a native method call, it will flush the values from object into the native memory and could accidentally nullify the change. When Structure's field has volatile modifier, JNA will not write the value to native memory, unless Structure.writeField() method is explicitly called [12]. Java's final modifier can be used to prevent Java code from updating Structure's field, but it can be modified in native code. Structure.read() method will still flush values from native memory to Java object [12], but it can be dangerous, for the JIT-compiler will assume that final fields can never change, and stale value could be read instead of the change made in the native code.

4 Foreign Functions and Memory API

JDK 22 release in 19th of March 2024 [2] introduced new Java API to the public, Foreign Functions and Memory API (FFM API) [13]. Being in development since Java 14, the API has been refined over the years, with collaboration with large projects like Netty, Apache Lucene, and Apache Tomcat. Motivation for developing another foreign function interface into core JDK along the side of JNI was to create cross-platform Java-only API. FFM API allows programmer to

- read and write native memory in safe manner.
- create native memory arenas.
- allocate and de-allocate native memory on demand.
- load and link native binaries.
- invoke foreign functions.

4.1 Memory

FFM API provides a way to allocate, de-allocate, and manipulate native memory on demand, with safety built in the API [13]. Memory can be tied to a lifetime and its layout can be modelled with declarative style, allowing the API conduct necessary safety checks to prevent memory issues like corruption, segmentation faults, and use-after-free bugs [13]. All of this can be done in Java code, using Java language features.

4.1.1 Memory Segment

FFM API abstracts the concept of contiguous region of memory as MemorySegment Java objects, with the actual backing memory residing on-heap or off-heap [14]. There are three types of MemorySegments [14], differentiated by backing the memory's location:

- native segment, backed by native off-heap memory.
- mapped segment, backed by a region of memory mapped off-heap memory.
- array segment, backed by an on-heap array, managed by garbage collector.

MemorySegment Java class does not branch out to separate child classes to model these types but offers methods to detect the type of the backing memory. This is because memory, be it on- or off-heap is still fundamentally the same concept. It is a contiguous region of bytes, and memory operations require the same steps in each case.

Every type of MemorySegment has an address, size boundaries and temporal boundaries [14]. The address can be thought of as the pointer to the start of the segment. Size boundaries determine the number of bytes that can be accessed from the beginning of the segment (address). Size boundaries together with the address allow runtime safety checks to be performed, preventing buffer overflow bugs [14]. Temporal boundaries determine the lifetime of the segment and are used to prevent access to the segment after its de-allocated, preventing use-after-free bugs [14].

The address of native segment is the actual address in process virtual memory address space. For on-heap array segments, the address refers to the offset from the start of the heap region containing the array. Because the backing array is managed by the garbage collector, the region can, and will be moved around the heap during the program execution, but the offset from the start of the region will always remain valid [14].

MemorySegments temporal boundaries tie the segment to a concept of memory scope, also referred as lifetime of the memory segment [15]. Lifetime can be unbound or bounded. Unbounded lifetime, or global scope, means the memory segment is always alive [15]. Global scoped segment can be acquired by:

- Allocating memory using global Arena
- Using raw pointer MemorySegment.ofAddress(long ptr, long size)
- Allocating zero sized segment, also known as void pointer

Bounded lifetime scope can be invalidated explicitly in Java code, or automatically by the garbage collector [15]. Automatic scoped segments can be acquired by:

- Allocating from automatic Arena
- Wrapping on-heap Java array or buffer into MemorySegment

MemorySegments can also limit multithreaded access. All on-heap MemorySegments can be accessed by any thread, but native segments are tied to the settings of the Arena they were allocated from [14].

4.1.2 Arena

All memory in FFM API is tied to a concept of memory arena. Memory arenas are used in regional memory management strategy, which uses often large memory regions, and provide slices of that region to the client code [16, p. 1]. The benefit of using memory arenas is reduced number of allocation system calls to the operating system, since use of arenas tend to over-allocate number of memory pages at the beginning. When the client request memory chunk, it is provided in the user space code, being more performant than making a call to the operating system every time more memory is required. Java Virtual Machines use this strategy when managing heap and allocating Java objects [17]. Another benefit is that when arena is no longer needed, it can be de-allocated all at once [16, p. 1].

In FFM API, memory arenas allocate or take ownership of MemorySegments [18]. An Arena has a scope, called the arena scope, which defines the lifetime and thread restrictions of its memory and all memory segments associated with it share that scope [18]. There are four types of arenas in Arena API, global, automatic, confined, and shared arena, with different attributes listed in table 4.

Table 4. Available types of memory arenas in FFM API [18].

Kind	Bounded lifetime	Explicitly closeable	Multithreaded
Global	No	No	Yes
Automatic	Yes	No	Yes
Confined	Yes	Yes	No
Shared	Yes	Yes	Yes

Segments allocated in the global arena remain alive the whole lifetime of the program and are accessible by any thread. Automatic arena hands the management of memory lifetime to the garbage collector [18]. It is important to note that for the automatic arena to de-allocate its memory, it and all the memory segments allocated from it must be eligible for garbage collection [18]. Listing 13. demonstrates usage of global and automatic arenas.

```
// Segment is always available during program
MemorySegment globalSegment = Arena.global().allocate(64);

MemorySegment managedSegment = Arena.ofAuto().allocate(64);
…
managedSegment = null;
// Segment will be de-allocated when garbage collection runs and no
// other references exist.
```

Listing 13. Global and Automatic arena allocate memory segments. Segment from automatic arena is eligible for de-allocation when garbage collection runs.

Memory allocated in confined and shared arenas is only de-allocated when the arena is closed, as demonstrated in listing 14. The difference is that segments from confined arena are guarded by strong thread-confinement and are only

accessible from the thread that owns the arena, usually being the thread that created it [18].

```
MemorySegment segment = null;
try (Arena arena = Arena.ofConfined()) {
    segment = arena.allocate(100);
    ...
} // segment region deallocated here
 segment.get(ValueLayout.JAVA_BYTE, 0); // throws IllegalStateException
```

Listing 14. Creation of confined arena in Java try-with-resources language feature. After the arena is closed, any access to memory segments associated results in IllegalStateException [18] .

Unlike confined arenas, memory from shared arenas is accessible for any application thread. A good example of a use case for shared arenas is when segments of memory need to be processed in parallel with Java Stream API, like demonstrated in listing 15.

```
try (Arena arena = Arena.ofShared()) {
     SequenceLayout SEQUENCE_LAYOUT = MemoryLayout
               .sequenceLayout(1024, ValueLayout.JAVA_INT);

     MemorySegment segment = arena.allocate(SEQUENCE_LAYOUT);
     int sum = segment.elements(ValueLayout.JAVA_INT)
              .parallel()
              .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
              .sum();
}
```

Listing 15. Example of shared arena usage. MemorySegment of 1024 32-bit integers is allocated from shared arena, and the values are summed up using parallel stream [14].

4.1.3 Data Alignment

By default, FFM API requires memory access to respect alignment constraint [14]. An address in physical memory is aligned according to layout if the address is a multiple of the constraint. For example, physical address 1000 is aligned to 8-byte, 4-byte and 2-byte constraints. Native segment with address of 1000 can be accessed at offsets 0, 8, 16, 24 and so on, under an 8-byte constraint, because the physical addresses of the offsets (1000, 1008, 1016, 1024) are 8-byte aligned [14]. To allocate native segment with address aligned to certain constraint, FFM API Arenas accept alignment argument.

Alignment constraints also apply to on-heap array segments. When the garbage collector moves arrays around the heap, it guarantees that the array's offset address is always align to the element type's byte alignment (Table 4). To guarantee aligned access, array segment can only be accessed with less than or equal alignment of the array's element's type [14].

Table 4. Primitive array alignments can be acquired by ValueLayout primitive constants. [14]

Array type	Maximum supported alignment (in bytes, platform dependent)
boolean[]	ValueLayout.JAVA_BOOLEAN.byteAlignment()
byte[]	ValueLayout.JAVA_BYTE.byteAlignment()
short[]	ValueLayout.JAVA_CHAR.byteAlignment()
int[]	ValueLayout.JAVA_SHORT.byteAlignment()
float[]	ValueLayout.JAVA_FLOAT.byteAlignment()
long[]	ValueLayout.JAVA_LONG.byteAlignment()
double[]	ValueLayout.JAVA_DOUBLE.byteAlignment()

4.1.4 Zero-length MemorySegments

FFM API uses so called "zero-length" MemorySegments to represent a foreign pointer. When interfacing with foreign functions that return a pointer, the only information Java runtime has is the value of the pointer itself. When translated to MemorySegment object, the pointer value is the address of the segment object, but the size is cannot be known [14]. It can be though as a void pointer in the context of C or C++, or java.lang.Object reference in Java, since it lacks any information of the type of data it points to. An attempt to access zero-length segment memory will throw IndexOutOfBoundsException.

Zero-length segment can be casted to a specified size, but it is inherently unsafe, because the actual size cannot be guaranteed to be correct by the FFM API. Re-interpretation methods in MemorySegment API are restricted in Java Platform, producing runtime warnings of its unsafe nature [14]. Runtime warnings can be turned off by adding JVM runtime argument --enable-native-access=M1,M2 where Mn are the names of modules that are allowed to make use of restricted API methods [19]. In listing 16, a zero-length segment “ptr” is created from native pointer, and then re-interpreted as an array of ten platform dependent sized integers. After this, it is safe to read values from the segment.

```
MemorySegment ptr = MemorySegment.ofAddress(nativePtr);
MemorySegment ints = ptr.reinterpret(10 * ValueLayout.JAVA_INT.byteSize());
for (int i = 0; i < 10; i++) {
    System.out.println(ints.getAtIndex(ValueLayout.JAVA_INT, 0));
}
```

Listing 16. Zero-length segment is created from native pointer and re-interpreted as an array of ten 32-bit integers.

4.1.5 MemoryLayout

FFM API uses MemoryLayout to describe the contents of the segment of memory or a slice of memory segment, and how it is interacted with [20]. To be able to work with raw memory, two concepts need to be defined: what are the size boundaries of the operation, and how the contents of the memory should be interpreted. FFM API enables modelling the memory access with MemoryLayout object [20]. MemoryLayout abstraction is branched out to two main categories: value layouts and padding layouts [20]. Value layouts model primitive datatypes by defining the size of the primitive’s value, and its type. Padding layout is an additional memory that is needed to ensure structured data is aligned accordingly.

Arrays of primitive values can be modelled with FFM API’s SequenceLayout interface [21]. SequenceLayout models repetitive sequence of defined element’s MemoryLayout, visualized in figure 3. Size of the SequenceLayout is the number of elements times the size of the element MemoryLayout size. The alignment of SequenceLayout is the element MemoryLayout byte size [21].

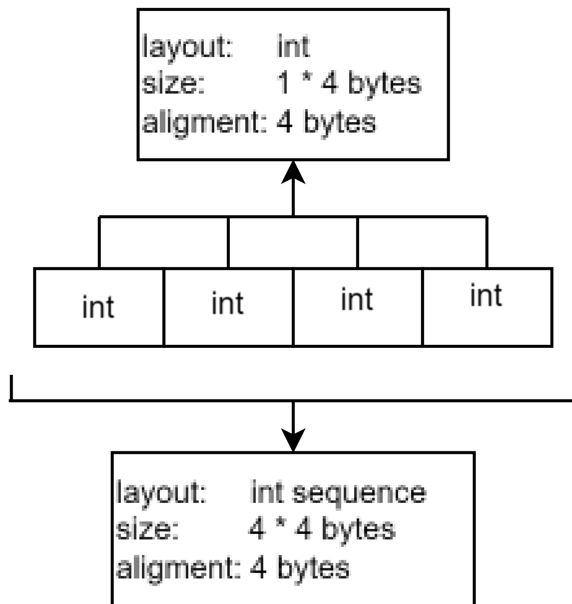

Figure 3. MemoryLayout of sequence of four 32-bit integers.

For structured data, FFM API supports aggregating primitive value layouts into larger constructs, modelling C type structs using StructLayout. Listing 17 declares C struct Point, with one char field "isValid" of size 1, and two int fields, each size 4.

```
typedef struct Point {
    char isValid;   // 1 byte
    int  x;         // 4 bytes
    int  y;         // 4 bytes
} Point;
```

Listing 17. C-style struct with 9 bytes of value size. The compiler will add padding after the char "isValid" field to ensure proper data alignment, resulting in actual size of 12 bytes, as seen in figure 4.

Not visible in the code, but C compiler will add 3 bytes of padding after the "isValid" field, to ensure data alignment of 4-bytes is respected. This is visualized in figure 4. When defining the memory layout with StructLayout, this padding must be explicitly added (listing 18). Listing 19 demonstrates access to named fields of the Point struct by using VarHandles.

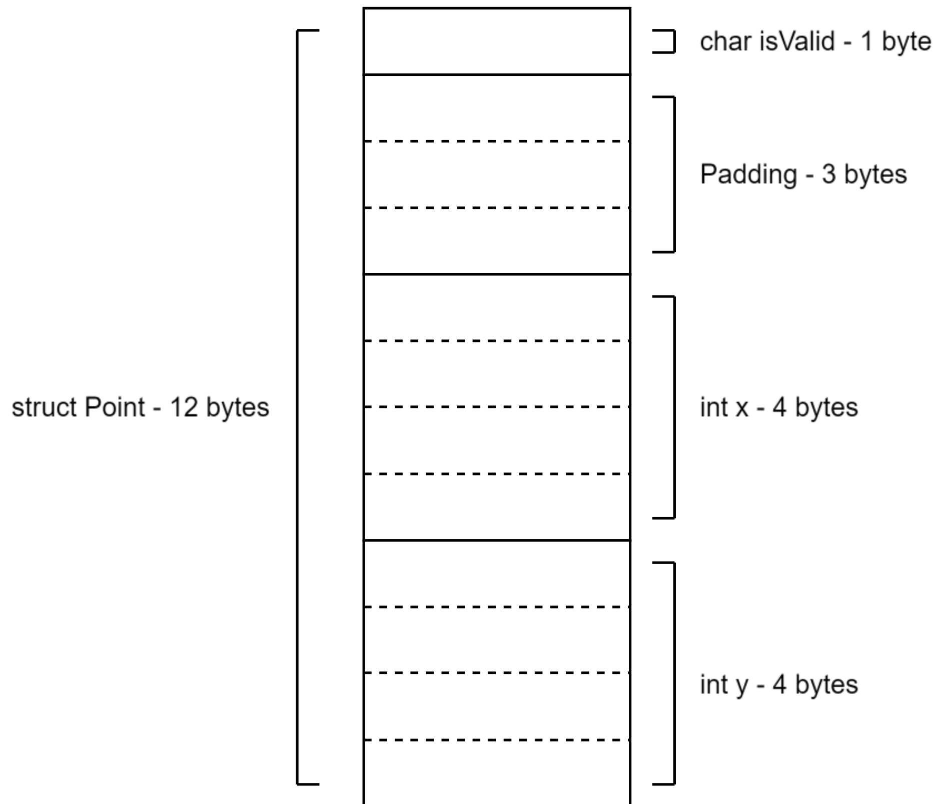

Figure 4. Memory layout of the Point struct after additional padding is added.

```
StructLayout point = MemoryLayout.structLayout(
    ValueLayout.JAVA_BYTE.withName("isValid"),
    MemoryLayout.paddingLayout(3),
    ValueLayout.JAVA_INT.withName("x"),
    ValueLayout.JAVA_INT.withName("y")
);

VarHandle isValidHandle = point.varHandle(PathElement.groupElement(isValid"));
VarHandle xHandle       = point.varHandle(PathElement.groupElement("x"));
VarHandle yHandle       = point.varHandle(PathElement.groupElement("y"));
```

Listing 18. StructLayout is created with same fields as in the native struct, but padding is added explicitly. VarHandles are generated for accessing structures fields by names.

```
try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment segment = offHeap.allocate(point);
    populateStruct(segment, isValidHandle, xHandle, yHandle);

    if ((byte)isValidHandle.get(segment, 0) == TRUE) {
        System.out.println("X: " + xHandle.get(segment, 0));
        System.out.println("Y: " + yHandle.get(segment, 0));
    }
}
```

Listing 19. MemorySegment is allocated from confined memory arena, and fields are accessed using VarHandles.

4.2 Foreign Functions

Foreign Functions API in FFM relies heavily on memory concepts explained in chapter 5.1. After loading a native library into memory, FFM API models function addresses as zero-length MemorySegments [13]. Native code is just instructions in memory, and all functions have a start address. Foreign Functions API classes SymbolLookup and Linker provide three ways of loading and locating native functions from libraries: library lookup, loader lookup, default lookup.

Library lookup loads library binary using provided library name into native memory. Repeating calls to load the same library get ignored [22]. Memory containing the library is associated with the Arena provided by caller, and the lifetime of the library is tied to the settings of the provided Arena. When the provided arena is closed, library gets unloaded [22], as demonstrated in Listing 20.

```
try (Arena arena = Arena.ofConfined()) {
     // libGL.so loaded here
     SymbolLookup libGL = SymbolLookup.libraryLookup("libGL.so", arena);
     MemorySegment glGetString = libGL.find("glGetString").orElseThrow();
     ...
 } //  libGL.so unloaded here
```

Listing 20. Native libGL-library is loaded by using confined Arena [22].

Loader lookup relies on native libraries already loaded with same method that is used with JNI library loading. Loader lookup expects library is already loaded with System.load(String) or System.loadLibrary(String). This method of loading will group up all symbols in different libraries together, and library name or path does not need to be specified, as seen in Listing 21. Libraries cannot by unloaded on demand, they are linked to the JVM class loader that loaded them.

```
System.loadLibrary("GL"); // libGL.so loaded here
...
SymbolLookup libGL = SymbolLookup.loaderLookup();
MemorySegment glGetString = libGL.find("glGetString").orElseThrow();
```

Listing 21. Same library as in Listing 20 is first loaded in JNI style, and FFM API symbol lookup expects to find it. [22]

Default lookup is performed by FFM API Linker class, and it can be used to find common functions associated with the platform. For example, on Linux platform, Linker implementation might find symbols, such as malloc and qsort from glibc library [22].

4.2.1 Linker

FFM API SymbolLookup provides the address of the native function, but that is not enough to make a native call. The function signature must be defined, and the call must be made according to the platform ABI. Linker.nativeLinker() method provides Linker implementation that is aware of the platform JVM is running on, and based on that, it can make the correct steps defined in ABI calling conventions [23]. With JNA, this was the responsibility of libffi library. With FFM API, if the Linker is not optimized for the platform, it delegates the job for libffi. Platforms supported by Linker directly [13] are:

- Linux/x64
- Linux/AArch64
- Linux/RISC-V
- Linux/PPC64
- Linux/s390
- macOS/x64
- macOS/AArch64
- Windows/x64
- Windows/AArch64
- AIX/ppc64

Linker takes the function address and function signature and creates callable MethodHandle Java object, as demonstrated in Listing 22. Function signature is defined as FunctionDescriptor, obtained by FunctionDescriptor.of() factory method, where first argument is the layout of return value and argument layouts are passed as variadic arguments.

```
Linker linker = Linker.nativeLinker();
MethodHandle strlen = linker.downcallHandle(
    linker.defaultLookup().find("strlen").get(),
    FunctionDescriptor.of(JAVA_LONG, ADDRESS)
);

try (Arena arena = Arena.ofConfined()) {
    MemorySegment str = arena.allocateFrom("Hello");
    long len          = (long)strlen.invoke(str);
}
```

Listing 22. C-standard library function strlen MethodHandle called with native string. [13]

If function has no return value, FunctionDescriptor.ofVoid() is used. User defined StructLayouts and UnionLayouts can be used in FunctionDescriptors.

5 Benchmarks

To test each method of interfacing with native libraries, a Java project was created with three main entities:

- Native test library – produce shared library for java implementations.
- Java implementations – explore different solutions.
- Benchmarks – test Java implementations.

The main project uses Gradle as main build tool, contains targets for building all the components and running tests and benchmarks. Gradle orchestrates other build tools for various parts of the project. Java implementations are built with Gradle, native C library with GNU make, and benchmarks are built using Maven.

5.1 Native test library

The native library being tested is a simple C library (Appendix 1) for indexing words in a text file. Library client can index words by file offsets and make queries for occurrences of the word in the text file with surrounding text as context. Queried word is normalized, discarding punctuation and case sensitivity. The caller provides a buffer where results are written. The index only

allows reading operations after it is created. If the indexed file changes, the whole index must be re-created.

The index data structure is implemented with a hash table of words mapped to word entry objects, holding the file offsets. Hash collisions are handled with a separate chaining method, using dynamic arrays of index entries. The hash function used is SDBM string hash function. The index data structure is not thread safe.

Word index is chosen to represent the native library in the benchmarks, because index creation can consume large amount of memory, and is computationally expensive, while only requiring one transition from Java code to native code. On the other hand, index queries are small and frequent operations, and the overhead for calling native library is expected to be a real factor in the benchmarks.

5.2 Java implementations

Java part of the project contains five different implementations for WordIndex Java interface (Appendix 2):

- JavaWordIndex - uses commonly known Java APIs
- BufferedJavaWordIndex - uses buffered file IO.
- JNIWordIndex - uses JNI native methods.
- JNAWordIndex - uses JNA Library.
- FFMWordIndex - uses FFM API.

JavaWordIndex was the first pure Java implementation. When running initial testing, it was quickly noticed that this implementation was extremely slow on indexing compared to native implementation. It uses RandomAccessFile.readLine() method for indexing the words in the target file, and stack profiling pointed out this method as the performance bottleneck. BufferedJavaWordIndex was then created, which manages reading in buffered manner with RandomAccessFile.read(byte[]). The first iteration was decided to

be kept in benchmarking process, to show performance differences between pure Java solutions. The reason pure Java implementations are included in benchmarks, is that they provide more context on the performance of the implementations using the native library.

WordIndex implementations using the native library function as a stateful object between Java code and the library code. JNIWordIndex utilizes JNI native methods that map to library's JNI bindings (Appendix 3), JNAWordIndex uses JNA Interface Mapping, and FFMWordIndex uses FFM API MethodHandles. JNI and JNA use direct ByteBuffers when making calls to native code, and FFM implementation allocates and de-allocates buffers using memory arenas.

5.3 Benchmarks

Creating correct benchmarks is hard, especially for JIT-compiled languages like Java. If not careful, JVMs aggressive runtime optimizations could notice that the results of the code being evaluated are not needed in the program and JVM could decide to skip the whole method, producing misleading results.

JMH (Java Microbenchmark Harness) makes this process easier, since it will manage JVM JIT warmups, and provides tools to deal with runtime optimizations like dead code elimination and constant folding. JMH also spins up a new JVM instance for each benchmark target to prevent targets from interfering with each other [24]. Still, the benchmarks and conclusions drawn based on them must always be taken with some skepticism. Even when using tools like JMH, unknown outside factors could affect the results.

In this project, JMH Maven archetype was embedded into the Gradle project as a submodule, and both Java WordIndex implementations and native library were provided as dependencies for benchmark module. Listing 23. shows hardware specifications for the machine running the benchmarks.

```
OS: Ubuntu 22.04.4 LTS x86_64
Kernel: 6.5.0-27-generic
CPU: AMD Ryzen 5 1600X (12) @ 4.000GHz
GPU: NVIDIA GeForce GTX 1660 SUPER
Memory: 2964MiB / 15913MiB
```

Listing 23. Hardware specifications for benchmark runner machine.

5.3.1 Cold start benchmark

Cold start benchmark measures how quickly implementations can process 4MB text file in a “single shot” mode, meaning the method is ran and measured only once. This means JVM’s JIT-compiler has limited time window to inline Java bytecode to machine code. As Table 5. shows, the pure Java implementation suffers heavily from inefficient file IO. Buffered Java implementation fares much better in terms of speed but is still only half as quick as its native counterparts. Java implementations have measured significantly higher in memory allocation rate, for the native versions also allocate off-heap memory, which is not measured by this profiler.

Table 5. Benchmark results for index creation without JIT warmup.

Implementation	Score	Error	Unit	Allocation rate	GC time
Java	4154	N/A	ms/op	142936320 B/op	32 ms
Java buffered	186	N/A	ms/op	114374792 B/op	14 ms
JNI	97	N/A	ms/op	4204160 B/op	0 ms
JNA	105	N/A	ms/op	4411616 B/op	0 ms
FFM API	97	N/A	ms/op	4362728 B/op	0 ms

5.3.2 Garbage collection pressure benchmark

With batch processing benchmark, task is to index 512 files, each 4MB in size. This benchmark aims to evaluate the effect of accumulative garbage collector pressure, and its effect on performance. The benchmark was run with Java heap capped to 512MB. The results (Table 6) proved initial hypotheses to be

incorrect, and the G1 GC can easily keep up with single thread constantly allocating objects. With large data size, Java implementation without buffered IO took 18 minutes to complete the task, and this benchmark ran over 6 hours in total because of it.

Table 6. Benchmark results for index bulk index creation.

Implementation	Score	Error	Unit	Allocation rate	GC time
Java	1101183	± 1502	ms/op	35489115219 B/op	13195 ms
Java buffered	27413	± 321	ms/op	28171934227B/op	5375 ms
JNI	22635	± 293	ms/op	33293 B/op	0 ms
JNA	23173	± 353	ms/op	156864 B/op	0 ms
FFM API	22918	± 230	ms/op	121685 B/op	0 ms

5.3.3 Throughput benchmark

Throughput benchmark takes populated index and makes word queries against it. The results are listed in table 7. With measuring query throughput, pure Java implementations are clear winners. Crossing the JVM – native code bridge has its cost, and it shows when making continuous hash lookups beyond the native boundary.

What is interesting is that JNI and JNA solutions spent so much time on garbage collections. This seems out of place, since the memory allocation rate does not explain what is causing the long pause times. Creating a new direct ByteBuffer for passing the word parameter to native code each call is the culprit for this oddity. The profiler used does not pick it up, since it is native memory, but garbage collector is still doing the de-allocation for direct buffers. FFM solution differs from JNI and JNA on this aspect, for it allocates native MemorySegment tied to an arena with a lifetime of the method doing the query, and because of that, de-allocation does not rely on the garbage collector.

Table 7. Benchmark results for index query throughput.

Implementation	Score	Error	Unit	Allocation rate	GC time
Java	154555	± 459	ops/s	474 B/op	66 ms
Java buffered	154868	± 459	ops/s	458 B/op	64 ms
JNI	117912	± 376	ops/s	400 B/op	10511 ms
JNA	86827	± 3344	ops/s	873 B/op	10475 ms
FFM API	127497	± 471	ops/s	696 B/op	71 ms

5.4 Analysis

The benchmarks measuring indexing performance showed that utilizing the native library in heavy processing task added meaningful improvements to execution speed and reduced garbage collection pressure. Pure Java solutions were both slower than native ones in this benchmark. In the single shot benchmark, Java implementation without buffered IO was 42 times slower than native solutions, and the one with buffered IO took roughly twice as much time to finish compared to native solutions. Still, optimizing the performance bottleneck brought pure Java implementation in reasonable distance from the native solutions. Different foreign function solutions did not show any meaningful differences in native function call overhead for indexing.

In throughput benchmark, pure Java implementations had the best performance. This benchmark measuring frequent native calls, JNA's internal usage of Java reflection started to be a real factor, and it was ~30% slower that JNI and FFM API solutions.

6 Summary and conclusions

As stated in introduction, the objective of this thesis was to conduct in-depth research on three popular solutions for interfacing with native libraries from Java code and detect situations where they could be best applied. The solutions under review were Java Native Interface, Java Native Access, and Foreign

Functions and Memory API. Theoretical research was supplemented by testing computational performance and memory footprint of each solution.

This thesis can be referred when deciding how to include native libraries into Java projects. Based on the conducted research and performance testing, recommendations for choosing foreign function solution for Java projects can be made.

Java Native interface is recommended when minimal overhead is required and Java 22 is not available, or interaction with JVM from native code is deciding factor. Java Native Access is recommended when call overhead is not a critical factor, and project can include 3rd party library. FFM API is recommended to be used in projects where Java 22 is available, and small overhead or low-level memory manipulation is desired.

It was also found that using native libraries is not always necessary for solving performance issues of Java applications. The first course of action is to optimize badly performing Java code and conduct benchmarks to see if acceptable speed or memory usage can be achieved without extracting the code to a native library.

References

1 Oracle. Java SE 22 Native Interface Specification. Available from: https://docs.oracle.com/en/java/javase/22/docs/specs/jni/ [Accessed 2nd April 2024]

2 Oracle. JDK Releases. Available from: https://www.java.com/releases [Accessed 2nd April 2024]

3 Lindholm, Tim; Yellin, Frank; Bracha, Gilad; Buckley, Alex; Buckley & Smith, Daniel. 2024. The Java Virtual Machine Specification Java SE 22 Edition. Oracle: Oracle America, Inc

4 Artima. Venners, Bill. Inside the Java Virtual Machine. Available from: https://www.artima.com/insidejvm/ed2/jvm9.html [Accessed 2nd April 2024]

5 Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, Alex, Smith, Daniel & Bierman, Gavin. 2024. The Java Language Specification Java SE 22 Edition. Oracle: Oracle America, Inc

6 JNA. GitHub. Available from: https://github.com/java-native-access/jna [Accessed 3rd April 2024]

7 JNA. GitHub. Functional Overview. Available from: https://github.com/java-native-access/jna/blob/master/www/FunctionalDescription.md [Accessed 3rd April 2024]

8 JNA. GitHub. FrequentlyAskedQuestions. Available from: https://github.com/java-native-access/jna/blob/master/www/FrequentlyAskedQuestions.md [Accessed 3rd April 2024]

9 JNA. GitHub. Direct Mapping. Available from: https://github.com/java-native-access/jna/blob/master/www/DirectMapping.md [Accessed 3rd April 2024]

10 JNA. GitHub. Getting Started. Available from: https://github.com/java-native-access/jna/blob/master/www/GettingStarted.md [Accessed 3rd April 2024]

11 JNA API 5.14.0 Overview. Available from: https://java-native-access.github.io/jna/5.14.0/javadoc [Accessed 3rd April 2024]

12 JNA API 5.14.0 Javadoc. Structure. Available from: https://java-native-access.github.io/jna/5.14.0/javadoc/com/sun/jna/Structure.html [Accessed 3rd April 2024]

13 Cimadamore, Maurizio. JEP 454: Foreign Function & Memory API. Available from: https://openjdk.org/jeps/454 [Accessed 5th April 2024]

14 FFM API Javadoc. Memory Segment. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/MemorySegment.html [Accessed 6th April 2024]

15 FFM API Javadoc. Memory Segment. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/MemorySegment.html [Accessed 6th April 2024]

16 Gay, Edward. 2001. Memory Management with Explicit Regions. University of California, Berkeley.

17 Java SE 21 Core Libraries. On-Heap and Off-Heap Memory. Available from https://docs.oracle.com/en/java/javase/21/core/heap-and-heap-memory.html [Accessed 6th April 2024]

18 FFM API Javadoc. Arena. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/Arena.html [Accessed 6th April 2024]

19 FFM API Package summary. Restricted methods. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/package-summary.html#restricted [Accessed 6th April 2024]

20 FFM API Javadoc. MemoryLayout. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/MemoryLayout.html [Accessed 7th April 2024]

21 FFM API Javadoc. SequenceLayout. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/SequenceLayout.html [Accessed 7th April 2024]

22 FFM API Javadoc. SymbolLookup. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/SymbolLookup.html [Accessed 7th April 2024]

23 FFM API Javadoc. Linker. Available from https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/Linker.html [Accessed 7th April 2024]

24 Java Microbenchmark Harness. JMH GitHub git repository. Available from https://github.com/openjdk/jmh [Accessed 10th April 2024]

C Header file of the native word index library

```
#if !defined(W_INDEX_H)
#define W_INDEX_H

#include <stdarg.h>
#include <stdbool.h>
#include <stdlib.h>
#include "wordindex/analyzers.h"
#include "wordindex/utils.h"

#define BUFF_TERM_MARK ((uint32_t)0)
#define BUFF_TERM_MARK_SIZE (sizeof(BUFF_TERM_MAKR_TYPE))

enum context { NO_CONTEXT = 0, SMALL_CONTEXT = 16,
               MEDIUM_CONTEXT = 64, LARGE_CONTEXT = 128};
typedef struct wordindex WordIndex;

/**
 * @brief Opens a WordIndex over specified file.
 *
 * @param file path        Path to text file to index
 * @param analyzer         Analyzer to use in indexing
 * @param capacity         Capacity estimate
 * @param read_buffer_size Size of the buffer used for reading the file
 * @param compact          Should index be compacted
 * @return WordIndex*      Pointer to index object
 */
WordIndex *file_word_index_open(const char *filepath,
    enum index_analyzer analyzer, size_t capacity, size_t read_buffer_size,
    bool compact);

/**
 * @brief Fills the provided buffer with results and returns opaque iterator
 *        pointer. When all results are read, returned ptr is NULL. To fill
 *        buffer with next results, pass the same iterator ptr.
 *
 * @param index            Index object
 * @param read_buffer      Buffer to read results to
 * @param read_buffer_size Size of the read buffer
 * @param word             Word to look for
 * @param context          Context size to include
 * @return void*           Iterator
 */
void *file_word_index_read_with_context_buffered(WordIndex *index,
    char *read_buffer, size_t read_buffer_size, const char *word,
    size_t word_len, size_t context, void *read_iterator);

/**
 * @brief Closes the index and frees all resources.
 * @param index Index object to close.
 */
void file_word_index_close(WordIndex *index);

/**
 * @brief Closes the iterator. Use if results still remain.
 * @param iter Iterator object.
 */
void file_word_index_close_iterator(struct index_read_iterator *iter);

#endif  // W_INDEX_H
```

WordIndex.java

```
package org.nse.thesis.wordindex;

import org.jetbrains.annotations.NotNull;
import java.io.FileNotFoundException;
import java.util.Collection;

/**Index that indexes words to their positions in the file. */
public interface WordIndex extends AutoCloseable {

    /**
     * Context used in {@link WordIndex} queries, to specify
     * the amount of leading and trailing bytes surrounding the
     * queried word.
     */
    enum ContextBytes {
        NO_CONTEXT,    // No context bytes.
        SMALL_CONTEXT, // 16 bytes of context.
        MEDIUM_CONTEXT,// 64 bytes of context.
        LARGE_CONTEXT; // 128 bytes of context.

        /**
         * @return Context in bytes.
         */
        public int size() {
            return switch (this) {
                case NO_CONTEXT -> 0;
                case SMALL_CONTEXT -> 16;
                case MEDIUM_CONTEXT -> 64;
                case LARGE_CONTEXT -> 128;
            };
        }
    }

    /**
     * Query the index for all occurrences of words from
     * indexed file, with specified on both sides of the word.
     *
     * @param word Word to search for.
     * @param ctx  The amount of context bytes to surround the word.
     * @return Collection of words with context.
     */
    @NotNull Collection<String> getWords(@NotNull String word,
                                         @NotNull ContextBytes ctx);

    /**
     * Queries the index with word with context, results
     * can be accessed through an iterator.
     *
     * @param word Word to search for.
     * @param ctx  The amount of context bytes to surround the word.
     *
     * @return Iterator that iterates over the results.
     * @throws FileNotFoundException If indexed file was deleted.
     */
    @NotNull WordContextIterator iterateWords(@NotNull String word,
                                              @NotNull ContextBytes ctx)
                                              throws FileNotFoundException;
}
```

JNI bindings embedded in wordindex C-library

```
#include <string.h>

#include "wordindex.h"
#include "wordindex/jni/org_nse_thesis_wordindex_jni_JNIWordIndexBindings.h"
#include "wordindex/utils.h"

JNIEXPORT jlong JNICALL
Java_org_nse_thesis_wordindex_jni_JNIWordIndexBindings_wordIndexOpen(
    JNIEnv *env, jclass class, jstring filepath, jint analyzer,
    jlong capacity, jlong bufferSize, jboolean compact) {

    const char *fpath = (*env)->GetStringUTFChars(env, filepath, NULL);

    WordIndex *index_handle = file_word_index_open(
        fpath, analyzer, capacity, bufferSize, compact);

    (*env)->ReleaseStringUTFChars(env, filepath, fpath);

    return (jlong)index_handle;
}

JNIEXPORT void JNICALL
Java_org_nse_thesis_wordindex_jni_JNIWordIndexBindings_wordIndexClose(
    JNIEnv *env, jclass class, jlong handle) {

    file_word_index_close((WordIndex *)handle);
}

JNIEXPORT jlong JNICALL
Java_org_nse_thesis_wordindex_jni_JNIWordIndexBindings_wordIndexReadWithContex
tBuffered(JNIEnv *env, jclass class, jlong handle, jobject jbytebyffer,
          jlong readBufferSize, jstring jword, jint word_len, jint context,
          jlong iter) {

    char *buffer = (*env)->GetDirectBufferAddress(env, jbytebyffer);
    WordIndex *index = (WordIndex *)handle;

    char *word = NULL;
    if ((void *)iter == NULL) {
        word = (char *)(*env)->GetStringUTFChars(env, jword, NULL);
    }

    void *result_iter = file_word_index_read_with_context_buffered(
        index, buffer, readBufferSize, word, word_len, context, (void *)iter);

    if (word != NULL) {
        (*env)->ReleaseStringUTFChars(env, jword, word);
    }

    return (jlong)result_iter;
}

JNIEXPORT void JNICALL
Java_org_nse_thesis_wordindex_jni_JNIWordIndexBindings_wordIndexCloseIterator(
    JNIEnv *env, jclass class, jlong iterator) {

    file_word_index_close_iterator((void *)iterator);
}
```