
University of Applied Sciences Coburg
Faculty of Electrical Engineering and Computer Science

Degree: Master of Computer Science

Master's Thesis

# Introduction, Performance Comparison and Optimisation of an ABI-Aware Approach to Access Structures in Java

Linus Andera

Date of Submission: 03.03.2026

Supervisor:
Prof. Dr. Florian Mittag, Hochschule Coburg

# Abstract (Deutsch)

Effizienter Zugriff auf native Strukturen (struct) ist wichtig für performancekritische Java-Anwendungen, wie beispielsweise Real-time rendering. Bestehende Bibliotheken – wie zum Beispiel die Lightweight Java Game Library (LWJGL), die Foreign Function & Memory API (FFMA) und Java Native Access (JNA) – ermöglichen diesen Zugriff. Allerdings bieten diese keine Unterstützung für Application Binary Interfaces (ABIs), wie zum Beispiel das Standard Uniform Block Layout (std140) von OpenGL, an. Deshalb stellt diese Arbeit einen ABI-bewussten Ansatz zum Zugriff auf native Strukturen in Java vor.

Zur Evaluation wurde eine Microbenchmark-Studie mithilfe von JMH durchgeführt, wodurch LUtils mit LWJGL, FFMA und JNA hinsichtlich Strukturerzeugung, Schreib-/Leseoperationen sowie Startzeitverhalten verglichen wird. Zusätzlich erfolgte eine Profiling-Analyse mit dem Async-Profiler, um Schwachstellen innerhalb LUtils zu identifizieren und Optimierungsmöglichkeiten abzuleiten.

Die Ergebnisse zeigen, dass LUtils aktuell nicht für performancekritische Anwendungen geeignet ist. Allerdings konnten potenzielle Optimierungen identifiziert werden, insbesondere in Bezug auf Java-Reflection-Nutzung, Allokationsstrategien sowie Schreib- und Leseoperationen.

# Abstract (English)

Efficient access to native structures is essential for performance-critical Java applications, like real-time rendering. Existing libraries - such as LWJGL, the Foreign Function & Memory API (FFMA) and JNA - facilitate this but do not provide support for Application Binary Interfaces (ABI) like OpenGL's Standard Uniform Block Layout (std140). Therefore, this thesis introduces an ABI-aware approach to access native structures in Java.

A microbenchmark study using JMH was conducted to compare LUtils with LWJGL, FFMA and JNA in terms of structure creation, write/read operations and startup performance. Furthermore, profiling was performed with the Async-Profiler to identify bottlenecks within LUtils and derive optimisations.

The evaluation reveals that LUtils – in its current form – should not be used for performance-critical applications. However, several possible optimisations regarding reflection usage, allocation strategies and write/read operations have been identified.

# Content

# Figures

# Tables

# Code Examples

# Abbreviations

| | |
|---|---|
| ABI | Application Binary Interface |
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| DLL | Dynamic Link Library |
| DSL | Domain Specific Language |
| FFMA | Foreign Functions and Memory API |
| GPU | Graphics Processing Unit |
| HTML | HyperText Markup Language |
| JDK | Java Development Kit |
| JFR | Java Flight Recorder |
| JIT | Just-In-Time |
| JMH | Java Microbenchmark Harness |
| JNA | Java Native Access |
| JNI | Java Native Interface |
| JSON | JavaScript Object Notation |
| JVM | Java Virtual Machine |
| LWJGL | Lightweight Java Game Library |
| PC | Program Counter |

# 1 Introduction

The high-level programming language Java does not support simple aggregate types such as C-style structures (`struct`) or unions [1]. Consequently, a variety of libraries have been developed to bridge this gap and enable interaction with native memory layouts [2]. This study explores a novel approach to dynamically create aggregate types from special Java classes at runtime, while providing support for different Application Binary Interfaces (ABI).

## 1.1 Context

When developing Java applications, it is sometimes required to interact with native functions, which can require structures or unions as parameters. The Vulkan API is an example which contains many methods with such parameters. Even the API entry point – `vkCreateInstance` – requires a pointer to a relatively complex `VkInstanceCreateInfo` structure [3]. That is why Java native interop libraries exist, which can generate Java glue classes directly from the C source code or the DLL files [2] (for examples see 3.2). These classes are used to create and edit the corresponding structures at runtime. However, these libraries work only with common C and C++ ABIs. The ABI defines rules for the structure's layout in memory. With these rules the structure's size, alignment and padding are calculated.

The ABI may differ for different processor architectures, operating systems and different use cases. For example, Windows requires a specific ABI convention for 64-bit systems[1]. However, challenges emerge when working with graphic processors and different graphics compute libraries. When passing structures to graphics card using compute libraries like Vulkan, OpenGL and OpenCL specific ABIs may be required[2,3] [4, S. 130, 161–163], which are not supported by current Java native interop libraries. That is why LUtils has been created, which allows the implementation of custom ABIs. This library generates the structure's memory layout at Runtime and allows the developer to specify an ABI explicitly.

## 1.2 Goals and Motivation

Given the problem discussed above, that ABIs required by Vulkan, OpenGL or OpenCL are not supported by current Java native interop solutions, LUtils has been developed and is introduced by this thesis. However, the new library has not seen much usage in real-world applications and is missing comparisons to already existing libraries in terms of performance and

[1] T. Whitney, x64 ABI conventions, Microsoft Learn, https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170
[2] https://docs.vulkan.org/spec/latest/chapters/interfaces.html#interfaces-resources-layout
[3] M. Segal and K. Akeley, The OpenGL® Graphics System: A Specification (Version 4.6), The Khronos Group, https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.withchanges.pdf

memory usage. Thus, it is unknown if this new approach is feasible for real-world applications. That is why the thesis seeks to:

- Compare the performance of LUtils against existing solutions – namely LWJGL, FFMA and JNA – in terms of structure creation, write/read operations, and startup costs. The goal of this comparison is to evaluate whether LUtils can be used in performance-critical applications or if other libraries should be preferred.
- Identify potential bottlenecks and areas for optimisation in the current implementation.
- Discuss the findings and possible optimisations.

Conversely, this study does not seek to discuss the useability of the examined libraries, nor does it attempt to analyse the trade-offs between flexibility and performance that arise when using a runtime-generated memory layout versus a precomputed one. Likewise, this thesis does not aim to analyse bottlenecks within other native interop libraries

To summarise, this thesis evaluates the proposed library – LUtils – by comparing it with established alternatives with respect to performance and Java-Heap memory consumption. The objective of this comparison is to determine whether LUtils is a viable solution for performance-critical applications, like real-time rendering systems. Furthermore, a detailed analysis of the benchmarks is intended to identify performance bottlenecks within LUtils and derive possible optimisations.

## 1.3 Task definition

To achieve the goals outlined above, several tasks must be carried out to evaluate the new library against existing solutions:

The first task is to identify suitable benchmarking methods and tools, which can accurately measure performance and memory usage. These tools must be able to measure execution time and analyse the Java heap memory.

The second task is to define experiments, which analyse structure creation, write/read operations and startup costs. Additionally, they must include different number of structures and structures with different sizes and complexities. Within the experiments structure creation time, write/read latency, and memory footprint must be measured.

After these experiments have been executed for both LUtils and the existing libraries – LWJGL, FFMA and JNA – the results are compared and discussed. It is expected that LUtils may introduce some performance overhead and higher memory usage compared to libraries with machine generated glue classes such as LWJGL. After the experiments the thesis aims to analyse bottlenecks within LUtils and derive potential optimisations.

The final step is to ensure reproducibility of the experiments. Therefore, the setup is documented and the code required to execute the experiments is provided, enabling others to verify the results.

## 1.4 Structure of the Thesis

The current chapter introduces the problem of working with native structures in Java and explains why different ABIs are required as well defining the goals and research questions of the thesis.

The following chapter presents technical foundations required to fully comprehend the thesis. This includes some special Java features, like reflection, the `Unsafe` class and the Java Native Interface (JNI) as well as the Application Binary Interface (ABI). Additionally, benchmarking and profiling concepts, including tools like JMH, JVisualVM, Async-Profiler and Java Flight Recorder are discussed.

Chapter 3 focuses on related work and existing technologies for Java native interoperability. It presents LWJGL, FFMA and JNA as well as introducing the newly developed library LUtils. Furthermore, implementation aspects such as structure definition, layout calculation and allocation strategy, are discussed for each library.

Three experiments are defined in chapter 4, including methodology, setup configurations and result evaluation. Each experiment evaluates the performance of the four libraries LWJGL, FFMA, LUtils and JNA with structures of different sizes and complexities - ranging from small to large and complex structures. Additionally, three benchmarks analysing structure creation, write/read operation and startup costs are defined per experiment. The results of each benchmark are interpreted based on comparisons to earlier benchmarks and the internal implementation of each library. The last section of this chapter concisely summarises the results and provides a performance overview across all libraries.

The experiments are followed by chapter 5 which performs an in-depth analysis using the Async-Profiler on the benchmarks executed for LUtils, thereby deriving possible optimisations to potentially increase runtime performance and decrease Java-Heap memory allocation overhead.

Finally, chapter 6 summarises the thesis and is followed by an overview of future research directions in chapter 7. These include a cross-platform evaluation of all four libraries, an analysis of the influence of partial write and read operations on performance, an in-depth analysis on one-time initialisations and the performance impact of different allocation strategies on complete applications as well as a re-evaluation of LUtils after implementing the proposed optimisations.

# 2 Fundamentals

This chapter introduces fundamentals necessary for a comprehensive understanding of the thesis and its interpretations. In particular, it outlines key Java mechanisms, including reflection, the `Unsafe` class and the Java Native Interface (JNI). Furthermore, the concept of the Application Binary Interface (ABI) is presented as it constitutes a central motivation for the development of LUtils. Finally, tools and problems related to benchmarking and profiling are presented.

## 2.1 Special Java Features

This section describes some special Java features like reflection, the `Unsafe` class and the Java Native Interface (JNI).

### 2.1.1 Java Reflection

Java reflection provides mechanism to inspect and manipulate classes, fields and methods dynamically at runtime [5], [6]. Through the `Class` API, information about a class can be accessed programmatically. For example, `Class.getFields()` returns all accessible public fields in a class, enabling libraries to analyse these fields at runtime. Fields can be written to and read from using the respective `Field.get()` and `Field.set()` methods. Additionally, annotations can be retrieved from classes, fields and methods using the `Field.getAnnotation()`, `Class.getAnnotation()` and `Method.getAnnotation()` methods respectively. Reflection also supports querying and invoking methods dynamically at runtime. However, while reflection offers flexibility at runtime, it introduces additional performance overhead.

### 2.1.2 Java Unsafe

The class `sun.misc.Unsafe` provides low-level operations that bypass many of Java's standard safety guarantees by enabling direct memory manipulation [7]. It allows reading from and writing to arbitrary memory locations using the `get*()` and `put*()` methods. For example, `Unsafe.getInt()` and `Unsafe.putInt()` can be used to read or write a 32-bit integer. Other methods provided by `Unsafe` include:

- `allocateMemory()`: Allocate native memory.
- `freeMemory()`: Free native memory previously allocated with `allocateMemory()`.
- `objectFieldOffset()`: Returns the memory offset of a given instance field within its containing object. This can be used together with `Unsafe.get*()` and `Unsafe.put*()` methods to directly access that field.
- `throwException()`: Throw any exception without enforcing Java's compile-time checked-exception rules. This means, that any throwable - including checked exceptions - can be thrown without being included in the method's `throws` clause.

However, since JDK version 9 the actual implementation of Unsafe has moved to `jdk.internal.misc.Unsafe`[4], which is not exported to the unnamed module and thus not accessible without adding the JVM options presented in Code 1.

```
--add-exports java.base/jdk.internal.misc=ALL-UNNAMED
--add-opens java.base/jdk.internal.misc=ALL-UNNAMED
```

Code 1: Access to `jdk.internal.misc`. JVM options to allow all the unnamed module to access the `jdk.internal.misc` package.`jdk.internal.misc.Unsafe`[5] provides more methods than `sun.misc.Unsafe`. For example:

- `put*Unaligned()`, `get*Unaligned()`: Unaligned memory access.
- `compareAndSet*()`: Atomically compare and write a value to an address.
- `allocateInstance()`: Allocate an instance of a specific class without executing its constructor.

[4] JEP 260: Encapsulate Most Internal APIs. https://openjdk.org/jeps/260
[5] https://github.com/openjdk/jdk/blob/cd50d78d447f9f39065bc844fb3041cba2db32db/src/java.base/share/classes/jdk/internal/misc/Unsafe.java

### 2.1.3 Java Native Interface (JNI)

The Java Native Interface (JNI) is a mechanism to integrate native code written in languages such as C or C++ within Java applications [1]. It enables Java code to invoke native functions and allows native code to access Java objects and methods. For example, a method declared as

```
public class MyClass {
    public static native int add(int a, int b);
}
```

Code 2: A native function in Java possible through the Java Native Interface (JNI).

can be implemented in C as

```
1  JNIEXPORT jint JNICALL Java_MyClass_add(JNIEnv* env, jobject obj,
2                                                      jint a, jint b) {
3   return a + b;
4  }
```

Code 3: The native implementation of the JNI function from Code 2.

## 2.2 Application Binary Interface (ABI)

The Application Binary Interface (ABI) defines a set of low-level rules that govern how structures are represented at the binary level [4], [8]. In particular, the ABI specifies the memory layout of structures, including the alignment requirements of primitive types, structures, unions and arrays, as well as the insertion of padding within structures to satisfy these alignment constraints. The ABI ensures that compiled code produced by different compilers or interpreters can interoperate correctly at the binary level. While the ABI additionally defines aspects such as register usage, calling conventions and stack layout, these topics are not relevant in the scope of this thesis and are therefore not discussed further.

To further illustrate why different ABIs may affect the structure layout, Fig. 1 describes the memory layout of a structure containing an int element and a 3-component float vector in memory using Microsoft's x64 ABI[6] and OpenGL's Standard Uniform Block Layout (std140)[7]. Noteworthy, is that – unlike std140 – x64 ABI does not specify vectors as special types. Instead, a float array of length three is used.

[6] T. Whitney, x64 ABI conventions. Microsoft Learn https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170

[7] M. Segal and K. Akeley, The OpenGL® Graphics System: A Specification (Version 4.6), The Khronos Group, https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.withchanges.pdf

| Address | x64 | OpenGL std140 |
|---|---|---|
| 0 | int | int |
| 4 | float3 | padding |
| 8 | float3 | padding |
| 12 | float3 | padding |
| 16 | | float3 |
| 20 | | float3 |
| 24 | | float3 |
| 28 | | padding |

Fig. 1: Memory layout of a structure across different ABIs. A structure with an int and a 3-component float vector in memory using different Application Binary Interfaces (ABI).

As seen in Fig. 1, the memory layout has major differences in padding, alignment and size. The x64 ABI aligns the float array to the next 4-byte boundary, whereas std140 aligns the 3-component vector to a 16-byte boundary, resulting in unused padding bytes. Consequently, reusing a host-side structure for GPU data transfer may produce incorrect results unless the layout rules of the target ABI are explicitly respected.

## 2.3 Benchmarking and Profiling

This section discusses problems related to benchmarking and profiling as well as discussing different benchmarking and profiling tools.

### 2.3.1 Benchmark Related Problems

While benchmarking Java-code some common pitfalls must be avoided to achieve accurate results. These issues are discussed below and are considered in the benchmarks presented in chapter 4 and 5.

*Dead Code Elimination*

Dead Code Elimination is a common issue in microbenchmarks [9], [10]. Benchmarks usually test a specific piece of code but leaving out the code consuming the computation result. However, the JIT compiler can detect code whose results are never used or whose execution has no effect and may therefore eliminate it entirely. This optimisation can unintentionally alter benchmark behaviour and lead to misleading results.

*Constant Folding*

Constant Folding describes the ability of the compiler to infer potentially complex calculations at compile time if they only depend on constants and replace its results with a constant value [9]. Thus, Constant Folding can incorrectly reduce benchmark performance time and produce misleading results.

*Non-representative Data/Use Case*

Another potential problem is to run the microbenchmark with non-representative data [9]. This may be data that is not used in production or data that does not cover a wide range of use cases. For example, testing structure creation only with very small structures, which leads to benchmark results that do not represent structure creation of larger structures.

*Wrong Stable State*

Microbenchmarks typically require multiple warmup iterations that execute the benchmark without contributing to the measured results [9]. These iterations allow the code to reach a stable state in which the JVM has completed its optimisations and performance has stabilised.

Furthermore, a wrong stable state can also be reached due to a benchmark design error. For example, if the memory is allocated only in the first iteration during the warmup and reused during all other iterations. Thus, not representing the time the benchmark took to allocate memory in the benchmark result.

However, when testing a cold execution of a benchmark it may be useful, that a stable state is not or only partly reached.

### 2.3.2 Profiling Related Problems

This subsection explains different problems regarding profilers. These characteristics are considered during the profiling of LUtils in chapter 5.

*The Safepoint Bias*

All sampling profilers must sample the program execution at random times [11]. To achieve random sampling the program may be interrupted every $t + r$ milliseconds, where $t$ is a fixed sample interval and $r$ is a random number between $-t$ and $t$. When profiling Java applications, the safepoint bias occurs when a profiler collects samples exclusively at JVM safepoints. This means that the profiler must wait until execution reaches the next yield point if it wishes to take a sample. These points are program locations where the JVM considers it safe to perform operations like garbage collection. Their placements are influenced by compiler optimisations. For example, if the compiler determines that a loop neither performs memory allocations nor

executes indefinitely, it may not place any yield points within that loop. Thus, a profiler that samples exclusively at these locations incorrectly attributes all time spent inside the loop to some line after the loop. Furthermore, JIT compilers may further adjust the placement of yield points due to other optimisations, such as inlining.

*Sampling and Inlining Skid*

As described above sampling profilers periodically record the program counter and look up the matching java method and byte code index [12], [13]. This progress introduces skid due to

- Non-uniform instruction cost: The program counter (PC) points to the next instruction to be executed. As a result, the cost of slow instructions is attributed to the following instruction instead.
- Pipelined, super scaler, out-of-order and speculative execution [14]: CPUs use pipelining to overlap instruction stages - fetching, decoding and executing multiple instructions simultaneously. Superscalar architectures allow multiple instructions to execute in parallel in the same cycle. Out-of-order execution lets the CPU execute instructions in a different order than specified by the program to maximise resource utilisation, while speculative execution predicts branch outcomes and executes instructions ahead of time. All of these optimisations effect execution time of many instructions, but the blame is always assigned to the instruction executed next (The PC).

The above skid only effects the translation from instruction to bytecode. That means that theoretically reading the method from the current Java Frame is the method which is to blame for the execution time. However, while this is correct for normal methods, inlined methods reintroduce the previous issue [13], because the current Java Frame corresponds to the last method that was not inlined. As a result, the PC must once again be mapped to a bytecode index, which must be associated with the appropriate inlined method. Furthermore, the problem is reinforced due to:

- The JIT-Compiler may reorder the byte code across inlined methods. This means lines of code from multiple different methods may be only instructions apart.
- The JIT-Compiler may identify and remove duplicated code across multiple inlined methods. That means that a single instruction may stem from multiple lines of code from different methods.

Overall, these skids imply that when many methods are inlined, the true performance cost may not originate from the method identified by the profiler, but rather from a different method – often one executed earlier [13].

### 2.3.3 Tools

This subsection introduces a wide range of Java related benchmark and profiling tools – namely JMH, JVisualVM, Async-Profiler and Java Flight Recorder. However, not all presented tools are used in the thesis. Instead, the goal of this section is to provide an overview of existing tools including their caveats, showing why some tools should not be used. Chapter 4 and 5 selects the tools used in the thesis.

*JMH*[8]

The Java Microbenchmark Harness (JMH) is an open-source tool to create microbenchmarks of Java code. Microbenchmarks measure the performance of small code fragments. By default, JMH measures only the execution time, but it can also report additional metrics, such as throughput, allocation rate and garbage collection information.

JMH allows single time and repeated execution of benchmarks. It also supports warmup iterations, enabling the code to reach a stable state before measurements are recorded. Although JMH provides many such features to help developers avoid common benchmark problems, it ultimately remains task of the developer to ensure correct benchmark code and execution. For example, parameters should be stored in non-final class variables to avoid constant folding and a `Blackhole` instance should be used to prevent the compiler from eliminating dead code. An example benchmark is displayed in Code 4.

```
1  @BenchmarkMode(Mode.AverageTime)
2  @State(Scope.Benchmark)
3  @Warmup(iterations = 5 , time = 10) // Warmup to reach stable state
4  @Measurement(iterations = 5 , time = 10)
5  @OutputTimeUnit(TimeUnit.NANOSECONDS)
6  public class ExampleBenchmark {
7      double x = 10.0; // Parameter in non-final class variable
8
9      @Benchmark
10     public double benchmark(Blackhole bh) {
11         bh.consume(Math.log(x));
12     }
13 }
```

Code 4: Example JMH benchmark avoiding common benchmark pitfalls, such as dead code elimination, constant folding and wrong stable state.

[8] https://github.com/openjdk/jmh

The `BenchmarkMode` annotation describes the measurement mode of the benchmark, including:

- `AverageTime`: Measures the average execution time
- `SingleShotTime`: Measure the time of a single execution
- `Throughput`: Measure the number of operations per unit of time
- `SampleTime`: Run multiple benchmarks and randomly sample the time needed for the operation

The `State` annotation enables the use of State variables (The variable `x` in Code 4) during execution. The scope of the state defines whether the state is shared between the whole benchmark or if each thread has its own state. The annotations `Warmup` and `Measurement` define how often the benchmark code is executed during warmup and measurement respectively. It defines the number of iterations and the time per iteration in seconds. For each iteration the benchmark code is executed until the time limit is reached. The `OutputTimeUnit` annotation defines the time unit used to display the measurement results.

To execute a benchmark JMH provides the `Runner` and `OptionsBuilder` classes. Using these, the developer can define configurations such as result location and format as well as attaching profilers and selecting which benchmarks are executed.

*JVisualVM*[9]

The Java VisualVM is an open-source monitoring and profiling tool [15], replacing the obsolete tool jconsole. VisualVM can attach to any running Java applications and analyse memory usage, CPU usage and thread activity. Additionally, it allows CPU time sampling. However, because it samples only at JVM yield points it is affected by the safepoint bias.

*Async-Profiler*[10]

Async-Profiler is a low-overhead sampling profiler that relies on an unofficial JVM API called `AsyncGetCallTrace` and the performance analysis tool perf [16]. Although all current OpenJDK distributions include the `AsyncGetCallTrace` interface, many other JDK implementations do not support it. Furthermore, the Async-Profiler is only available on Linux, asperf requires a Linux-based operating system.

[9] https://visualvm.github.io/
[10] https://github.com/async-profiler/async-profiler

The native method `AsyncGetCallTrace` retrieves the Java call trace of a specified thread asynchronously. By periodically calling this method the Async-Profiler samples CPU time. Since `AsyncGetCallTrace` does not rely on yield points it avoids the safepoint bias. However, the problems related to inlining and sampling skid persist.

Furthermore, to achieve accurate results the async-profiler requires the JVM flags `XX:+UnlockDiagnosticVMOptions` and `XX:+DebugNonSafepoints` as well as two Linux-kernel parameters (Code 5), which allow the capturing of kernel call stacks from a non-root process.

```
sysctl kernel.perf_event_paranoid=1
sysctl kernel.kptr_restrict=0
```

Code 5: Kernel parameters required by the Async-Profiler to capture kernel call stacks.

The Async-Profiler can be attached to a benchmark using JMH's `OptionsBuilder`. After the benchmarks have executed the profiling results are collected and the Async-Profiler generates an interactable flamegraph as HTML as seen in Attachment A 1. The colours indicate whether the code is inlined Java-code (blue), native-code (red) or standard JIT-compiled Java-code (green). Other colours may appear in the visualisation which are documented in the official reference materials[11].

*Java Flight Recorder (JFR)*

The Java Flight Recorder is sampling profiler, which uses code independent of the unofficial `AsyncGetCallTrace` API. It can access the call trace while mostly avoiding the safepoint bias if the `-XX:+DebugNonSafepoints` JVM flag is enabled. The Java Flight Recorder creates a .jfr file, which can be analysed in the tool Java Mission Control[12] or converted to a flamegraph HTML using a converter provided by the Async-Profiler.

Like the Async-Profiler it requires the JVM flags `XX:+UnlockDiagnosticVMOptions` and `XX:+DebugNonSafepoints` to achieve accurate results and can be attached to a JMH benchmark using the `OptionsBuilder`.

[11] https://github.com/async-profiler/async-profiler/blob/b55cb7c97392e41513f67a6212d39b9b2ecd7180/docs/FlamegraphInterpretation.md#understanding-flamegraph-colors
[12] https://github.com/openjdk/jmc

# 3 Related Work and Current Technologies

This chapter reviews existing work related to this thesis and presents current technologies for accessing native structures in Java. It begins with an overview of related research, followed by a concise introduction to established Java native access libraries – namely LWJGL, FFMA and JNA. Subsequently, the newly developed library LUtils is presented. Finally, the chapter concludes with a discussion of the internal implementation approaches of all four libraries, which form the basis required for many interpretations made in chapter 4.

## 3.1 Related Work

There is limited prior work addressing the specific focus of this thesis. One remotely related approach [17] aims to improve memory performance of embedded Java applications. It discusses dynamically changing the memory layout of arrays on embedded systems to improve performance. Another related study [18] discusses and compares tools that asses ABI compatibility. These tools detect potential bugs arising from ABI mismatches. Furthermore, the work highlights reasons why ABI compatibility can be problematic and how an understanding of ABIs might be required to solve ABI related problems.

Other research [19], [20], [21] discusses common practices and security related issues of the Java Native Interface (JNI). Another study [22] evaluates the performance of the Java Vector API in vector embedding operations, comparing it to pure Java solutions and C++ implementations called from Java.

The work most closely related to this thesis is the bachelor's thesis by Niklas Seppälä [23], which investigates whether Java native interop can improve performance compared to purely Java-based implementations. That study focuses on JNI, JNA and FFMA, concluding that utilising Java native interop is most suitable for larger tasks, which minimise communication between Java and native code. This represents the key difference to the approach presented by the present thesis, which aims to analyse the impact of creating and accessing structures in Java code without passing the structures to native code.

## 3.2 Current Technologies

Interfacing Java with native code is essential for many applications, therefore several libraries exist to facilitate this interaction. This section provides an overview of these libraries while focusing on structures and the Application Binary Interface (ABI).

Besides LUtils this thesis focuses on JNA, LWJGL and FFMA. However other libraries exist as well. For example, JavaCPP[13], SWIG[14] and JNIWrapper[15]. Compared to other technologies, LUtils is most similar to JNA, with the key difference, that the ABI can be selected at runtime and the ability to implement custom ABIs. The remainder of this section promptly introduces the Java native access technologies LWJGL, FFMA and JNA.

*Lightweight Java Game Library (LWJGL)[16]*

The Lightweight Java Game Library (LWJGL) provides a wide range of features for game development, including pre-generated native interop bindings. These bindings are created using an automated code generator that translates structure definitions, function signatures, and constants specified in a Kotlin-based domain-specific language (DSL) into corresponding Java interop code. For each native structure, a dedicated Java class is generated. However, this generator is intended for internal use within LWJGL and cannot easily be employed by developers to define custom structures without considerable setup effort.

Despite this limitation, LWJGL is included in this thesis because it is already used in real-world applications, particularly in the performance-critical real-time rendering of video games. Therefore, it serves as a relevant reference point for performance comparison.

It should also be noted that LWJGL targets only the x64 ABI, and the ABI cannot be selected or configured dynamically at runtime.

*Foreign Functions and Memory API (FFMA)[17]*

The Foreign Functions and Memory API (FFMA) is a base Java interface provided by Project Panama. It enables direct memory access and native function calls without requiring JNI glue code. Instead of requiring generated or handwritten java classes for structure representations, FFMA allows developers to define and manipulate native structures dynamically at runtime. However, FFMA does not calculate the memory layout of structures. Instead, the layout must be specifically defined in code including padding and alignment. In combination with the tool jextract, the Java code can be automatically generated from C or CPP Headers, but it does not provide the ability to specify different ABIs.

[13] https://github.com/bytedeco/javacpp
[14] https://www.swig.org/
[15] https://github.com/combit/JNIWrapper
[16] https://github.com/LWJGL/lwjgl3/tree/master/modules#generator
[17] https://openjdk.org/projects/panama/

*Java Native Access (JNA)*[18]

The library Java Native Access (JNA) enables calling native functions and creating structures with pure Java code. It uses prewritten JNI code to dynamically resolve and invoke native functions at runtime. It also computes native structure layouts by reading handwritten classes using reflection and determining their memory layout during execution. However, JNA does not offer ABI selection to developers.

[18] https://github.com/java-native-access/jna

## 3.3 LUtils

After some established native interop libraries have been presented, this section introduces the new library LUtils, highlighting its classes and explaining how memory is allocated, how the structure layout is calculated and how custom structures are defined. A general overview of the most important structure related classes is visible in Fig. 2. The class `Structure` is the main class from which every structure class must extend from. The classes like `BBInt1` and `BBFloat1` are wrapper classes for the respective primitive types. Furthermore, multi-component vectors such as `BBInt2` and `BBFloat4` exist as well. The `ComplexStructure` provides the developer with a way to implement custom structures with one or more child elements. Additionally, arrays of any structure class can be created using the `StructureArray` class. However, arrays of primitive types have additional implementations (e.g. `NativeInt32Array`) which provide performance benefits compared to `StructureArray`.

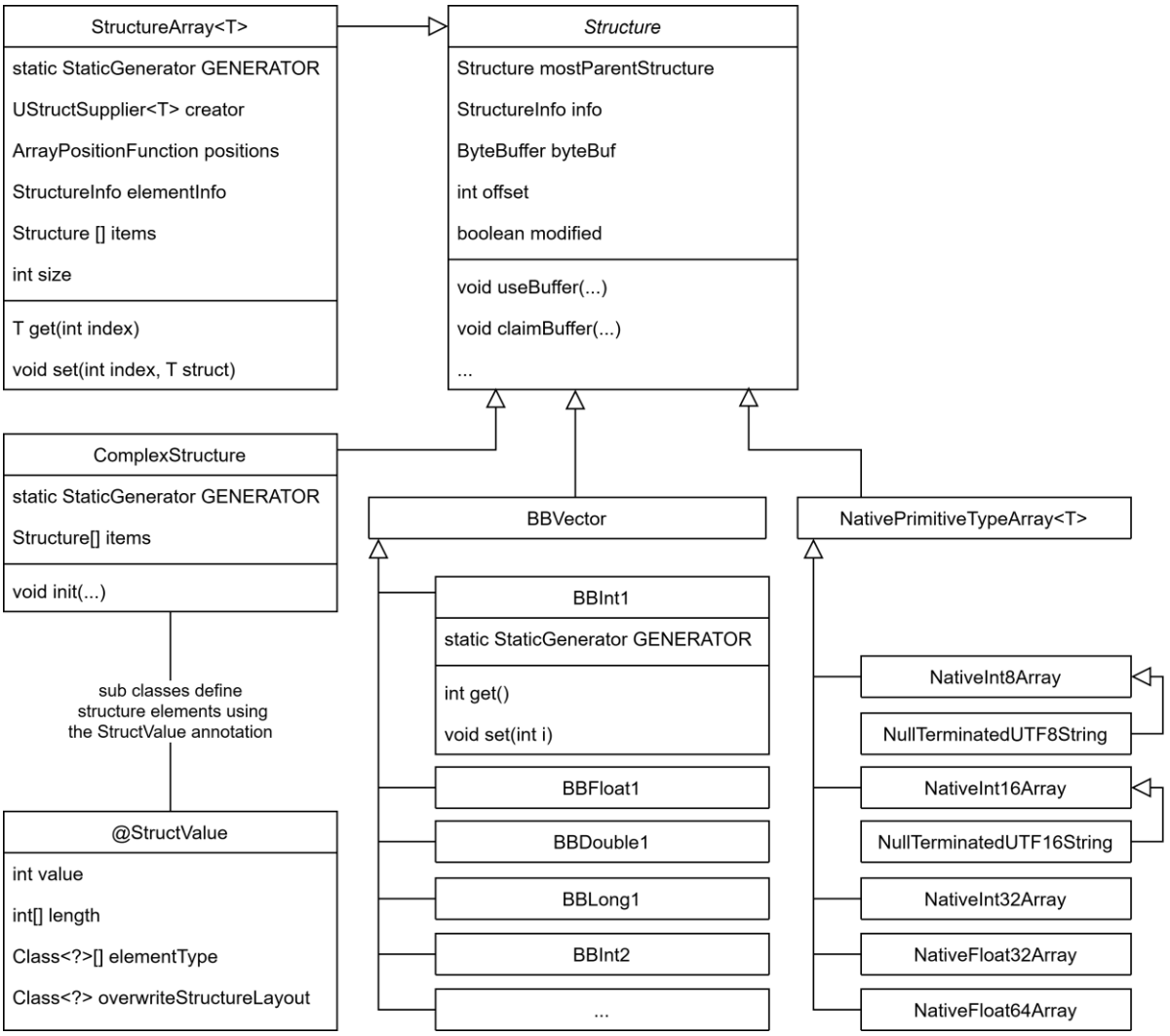


Fig. 2: Class diagram of LUtils showing the most important classes related to structures.

Each instance of the Structure class contains a `ByteBuffer` representing its native memory and a `StructureInfo`, which contains information about the structure's memory layout. However, a new structure instance can be created in three different states:

- *Unallocated*: The structure did not generate a `StructureInfo`. Some structures can generate their info in this state (e.g. `ComplexStructure`) and some cannot (e.g. `StructureArray`). This state is usually used if the structure is used as a child element in another structure. Thus, the parent structure must provide the required information to generate a `StructureInfo`.
- *Allocatable*: The structure has already generated a `StructureInfo` and is ready to claim a buffer using `claimBuffer()` or `allocate()`. This state is usually used if the structure is not used as a child element in another structure.
- *Allocated*: The structure has already generated a `StructureInfo` and `allocate()` has been called. This state is purely a quality-of-life feature. It is exactly the same as creating an allocatable structure and calling `allocate()`.

The information a structure class requires to generate a `StructureInfo` depends on the structure itself. Most structures only require an ABI. However, some structures require additional information like a length or an element type (e.g. `StructureArray`). Each structure class can define its requirements using the `StructureSettings` annotation. Which contains the following methods:

- `requiresCalculateInfoMethod`: Boolean, which defines how the `StructureInfo` for this structure class is generated. If it set to `true` the structure class must provide a `public static final StaticGenerator GENERATOR` variable which provides a `calculateInfo` method. If it is set to `false`, the structure class must provide a `public static final StructureInfo INFO` variable. Meaning that, in the latter case the `StructureInfo` for this structure is constant and does not depend on anything like ABI, length or element types.
- `customLengthOption`: This can be set to *not supported*, *optional* or *required*. It defines whether a length (for example array length) is required or optionally supported.
- `customElementTypesOption`: This can be set to *not supported*, *optional* or *required*. It defines whether an element type is required or optionally supported.
- `customLayoutOption`: This can be set to *not supported*, *optional* or *required*. It defines whether the ABI can (or must be) overwritten.

After explaining the core principles of LUtils, the following sub section highlights the classes `ComplexStructure` and `StructureArray` specifically. Furthermore, other features like modification tracking and ABI overwrite are introduced.

### 3.3.1 ComplexStructure

The `ComplexStructure` class can be used to implement custom structures, by extending this class and writing all structure elements as java class variables. For example, the structured defined by C-code in Code 6 can be defined in LUtils as seen in Code 7.

```
1  typedef struct SmallTestStruct2 {
2      int32_t aInt;
3      int8_t aByte;
4      int64_t aLong;
5  } SmallTestStruct2;
```

Code 6: An example structure defined in C.

The LUtils code is more complex, because every element is represented by a java variable initialised with an unallocated structure. Furthermore, every element must be annotated with the `StructValue` annotation.

```
1  public class SmallTestStruct2 extends ComplexStructure {
1    public final @StructValue(0) BBInt1 aInt = BBInt1.newUnallocated();
2    public final @StructValue(1) BBByte1 aByte = BBByte1.newUnallocated();
3    public final @StructValue(2) BBLong1 aLong = BBLong1.newUnallocated();
4
5    public SmallTestStruct2() { super(false); }
6  }
```

Code 7: An equivalent to the structure in Code 6 defined in LUtils.

The `StructValue` annotations signal LUtils that the variables are structure elements and optionally specify the position inside the structure. If the position of the elements is not specified, the order returned from `Class.getFields()`[19] is used instead. Based on testing this is always the order defined in the Java code. However, the Java documentation specifically states that the returned list is not in specific order. Thus, this should not be relied on. Instead, if a specific order is required it must be specified inside the `StructValue` annotation. If the classes extending `ComplexStructure` are machine generated, a more complex constructor is preferred. As example a generated constructor for `SmallTestStruct2` is displayed in Code 8.

[19] https://github.com/openjdk/jdk/blob/ffb6279c885e9d9a1a53ce7657390e286136c4b7/src/java.base/share/classes/java/lang/Class.java#L1918C21-L1918C26

```
1  public SmallTestStruct2(
2      @Nullable StructValue structValue,
3      boolean generateInfo
4  ) {
5          super(false);
6          init(structValue, generateInfo, aInt, aByte, aLong);
7  }
```

Code 8: Machine generated constructor of an LUtils structure. If the structure classes extending `ComplexStructure` are machine generated instead of handwritten, a more complex constructor is preferred. For example, the generated constructor for the structure defined in Code 7 is shown in this code example.

Noteworthy is the call to the `init()` function. The first parameter `structValue` can be used to change the ABI, by setting the value of `overwriteStructureLayout` to the desired ABI. The second parameter `generateInfo` can be used to already generate the `StructureInfo` for this instance. If `true` is passed, the instance is created in the allocatable state instead of the unallocated state. The remaining parameters are the structure elements in the same order as specified by their respective `StructValue` annotations. By passing these to the `init()` function, they do not have to be retrieved using reflection later.

### 3.3.2 StructureArray

The `StuctureArray` class can be used to create arrays of any other structure. For example, a structure containing an array can be created as seen in Code 9. This time the `StructValue` annotation is used to define the length and the element type of the array.

```
1  public class StructWithArray extends ComplexStructure {
2
3      @StructValue(value = 0, length = 20, elementType = SmallTestStruct2.class)
4      public final StructureArray<SmallTestStruct2> smallTestStruct3Array
5                      = StructureArray.newUnallocated(false, SmallTestStruct2::new);
6
7      public StructWithArray() { super(false); }
8  }
```

Code 9: A structure containing an array defined in LUtils. A structure defined in LUtils containing an array of 20 `SmallTestStruct2` structures.

### 3.3.3 Modification Tracking

Another feature LUtils provides is tracking where a structure has been modified. This feature is useful if a large structure must be uploaded to the GPU, but uploading the whole structure every time might be too performance heavy.

### 3.3.4 ABI Overwrite

All structures contained in LUtils itself support ABI overwrite. The ABI can be overwritten by passing a `StructValue` - with `overwriteStructureLayout` set to the desired ABI - to the static method `newAllocatable` of that structure. The ABI of classes extending `ComplexStructure` can be defined by annotating the class itself with the `StructureLayoutSettings` annotation. An example is displayed in Code 10.

```
1  @StructureLayoutSettings(DefaultABIs.MSVC_X64)
2  public class SmallTestStruct2 extends ComplexStructure {
3
4      public final @StructValue(0) BBInt1 aInt = BBInt1.newUnallocated();
5      …
6  }
```

Code 10: ABI Overwrite example. An example structure extending `ComplexStructure` with the ABI explicitly specified.

## 3.4 Internal Implementation of LWJGL, FFMA, JNA and LUtils

The goal of this section is to provide a short implementation analysis of the four libraries - LWJGL, FFMA, JNA and LUtils. The analysis focuses on the following aspects:

- Structure definition
- Structure layout calculation
- Reflection usage
- Allocator
- Write and read operations

These implementation details provide the necessary context for interpreting the benchmark results presented in the following chapter.

### 3.4.1 LWJGL

LWJGL's structures are defined by machine generated classes extending `org.lwjgl.system.Struct`[20]. The structure layout is stored in static final variables and is calculated during class loading. No reflection is used during layout calculation.

[20] https://github.com/LWJGL/lwjgl3

LWJGL provides multiple allocators[21] – namely malloc from stdlib, rpmalloc and jemalloc. The latter is used by default. The memory must be explicitly allocated and freed using `org.lwjgl.system.MemoryUtil`. In distinction to the other libraries, the memory is not initialised with zeros after allocating. While this provides better performance, the developer must clear the memory explicitly if zero-filled memory is required.

In addition to the allocators mentioned above, LWJGL provides the ability to allocate using `ByteBuffer.allocateDirect()` as well as a stack allocator, which allocates and manages a large chunk of native memory. Memory allocated by a stack allocator can be used by structures without requiring repeated allocations.

Furthermore, the structure classes generated by LWJGL contain methods to perform write and read operations for each structure field. These methods translate directly to `Unsafe.get*()` and `Unsafe.put*()`.

### 3.4.2 FFMA

FFMA's structures are not defined by a handwritten or generated class. Instead, an instance of `StructLayout` must be created using the `MemoryLayout.structLayout()` method[22]. The arguments of this method define the type and name of the structure elements. While this method validates the alignment of each member, it does not automatically add padding. Instead padding must be explicitly added using `MemoryLayout.paddingLayout(size)`. Consequently, the layout does not require calculation and can be stored by the developer for repeated usage. No reflection is used during this process.

FFMA provides its own memory management abstraction in the form of the `Arena` class, which offers a set of methods for allocating native memory. Memory allocated through an `Arena` is represented by instances of `MemorySegment`, which encapsulates the address and the size of the allocated memory. Internally, `Arena` relies on `Unsafe.allocateMemory()`. All native memory allocated by a specific `Arena` is released when `Arena.close()` is called.

Writing and reading operations on the native memory is achieved using the `VarHandle` class. For each structure element a `VarHandle` can be acquired by calling the `varHandle()` method on the `StructLayout` previously created. The `VarHandle` class provides `set()` and `get()` methods to write to and read from the native memory. Both methods are intrinsic candidates and have a polymorphic signature. Intrinsic candidates may be replaced by handwritten

[21] Tsakpinis, I. Memory management in LWJGL. https://blog.lwjgl.org/memory-management-in-lwjgl-3/ (2016)
[22] JEP 454: Foreign Function & Memory API. https://openjdk.org/jeps/454

assembly or bytecode to increase performance [15]. The signature of a polymorphic method is not defined by the method definition, but instead by the types at the call site[23]. Thus, the code executed after calling `VarHandle.set()` or `VarHandle.get()` is not deterministic cannot be derived from the Java source code.

### 3.4.3 JNA

A structure in JNA is defined by a handwritten class extending `com.sun.jna.Structure`[24]. The structure layout calculation happens the first time the constructor is executed, and the resulting layout is stored into a hash map. During layout calculation all fields of the handwritten class are iterated and validated using reflection.

Memory allocation happens automatically during constructor execution using the C-function malloc. JNA also registers a `Cleaner` to free the memory once it is no longer used.

Write and read operations are not performed directly on native memory. Instead, structure fields are represented as regular Java primitive fields, and write operations initially modify these fields through standard Java assignments. Only once the `write()` method is invoked, the current values of the Java fields are accessed – using reflection – and subsequently converted and written to the corresponding locations in native memory using custom JNI methods. Read operations follow the inverse process. Upon invoking the `read()` method, the native memory is read using custom JNI methods, converted to their respective Java types, and stored in the associated Java fields. The values can then be accessed through these fields in subsequent Java code.

### 3.4.4 LUtils

LUtils defines structures by extending the `ComplexStructure` class as described in 3.3.1. The structure layout calculation happens the first time a structure type in combination with a specific ABI is allocated, and the resulting layout is stored into a hash map. During layout calculation all fields of the handwritten class are iterated and their annotations read using reflection.

Memory allocation happens explicitly by calling the `allocate()` method and uses `ByteBuffer.allocateDirect()` internally.

Write and read operations are performed using methods provided by the wrapper classes (`BBInt1`, `BBFloat1`, …) and translate directly to `ByteBuffer.get*()` and `ByteBuffer.put*()`.

[23] https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html#sigpoly
[24] https://github.com/java-native-access/jna

# 4 Experiments

This chapter describes three different experiments comparing execution time and allocation rate of multiple benchmarks. To conduct these measurements, the Java Microbenchmark Harness (JMH) – presented in section 2.3.3 – is used. Each experiment compares structures of different sizes using three JMH benchmarks, which are executed for each of the four libraries separately:

Benchmark 1: Allocating structures

Benchmark 2: Allocate one or more structures as well as writing to and reading from the structures

Benchmark 3: Allocate one or more structures as well as writing to and reading from the structures (without warmup)

Benchmark 1 aims to compare the native memory allocation and initialisation of classes required for the structures. No write or read operations are performed on any structure elements. The benchmark is intentionally minimalistic to isolate overhead associated with memory allocation and structure creation.

The goal of benchmark 2 is to compare the performance of writing to and reading from structures. At least one structure is allocated, and all fields are set to random values. The random values are the same for each of the four libraries. They are generated using `java.lang.Random` with a fixed seed to allow reproducibility. Afterwards all fields are read and compared with the original input. The comparison code is the same for all four libraries, thus not skewing the benchmark results. Additionally, a reference benchmark is run which allocates the same structures but without performing any write or read operations. This reference benchmark is used to draw a reference line in the resulting plots, thereby isolating the overhead introduced by write/read access.

The goal of benchmark 3 is to compare the startup time across all evaluated libraries. However, it does not provide much valuable information in real use case scenarios, because the benchmark cannot differentiate between time spent in code executed once for the entire application and time spent in code executed once for every structure. Additionally, due to the way the benchmark is executed it cannot be analysed using the Async-Profiler. The reference benchmark executes the same code without performing one-time initialisations, therefore isolating the startup costs.

All benchmarks are executed on the same hardware, which is a Dell Latitude 5300 laptop with an Intel Core i5 vPro CPU with 16 Gigabyte of DDR4 RAM and no dedicated GPU. The laptop

is running the operating system Pop!_OS 24.04 LTS. Additionally, Intel SpeedStep, Intel Turbo Boost, hyper-threading and address space layout randomisation has been disabled to avoid performance fluctuations caused by these features [24]. Furthermore, the available memory has been fixed to 8GB using Java's `Xmx` flag.

Before discussing the experiments, the general experiment setup is shown. Then each experiment is described including the results and an interpretation of the results. The specific code of each experiment is available on GitHub[25].

## 4.1 Experiment Execution

The benchmarks are not run inside the IDE to avoid any skew. Instead, all benchmarks are run from a fat-jar, which bundles JMH and all necessary dependencies. The Jar contains a custom main method, which accepts three command line parameters:

- Benchmark identifier: The keyword for the native access library and the name of the benchmark class.
- Output directory: The directory in which the benchmark results will be stored.
- Async-Profiler: An optional boolean, whether to run the Async-Profiler alongside the benchmark. Defaults to `false` if this parameter is not present.
- Profile-Allocations: An optional boolean, whether to profile allocation rate instead of execution time. Defaults to `false` if this parameter is not present.

This means the jar is executed using a command as seen in Code 11.

```
Java -jar benchmark-runner.jar "lwjgl.Benchmark2" "2025-11-20_17-40-34"
```

Code 11: Benchmark execution command. Example command to execute "Benchmark2" for the native access library "lwjgl".

The custom main method runs the given benchmark using the JMH `Runner` and `OptionsBuilder` classes. It adds the `GCProfiler` to the benchmark, which measures Java heap memory allocation rate, garbage collection count and garbage collection time. Additionally, the result format is set to `JSON`. Thus, a JSON-file, containing all results is created once the benchmark has finished.

[25] https://github.com/lni-dev/thesis-lutils

Each benchmark exists for every native access library. The native access libraries are represented by the following keywords:

- lutils: The new library introduced by this thesis
- ffma: Foreign Functions and Memory API
- jna: Java Native Access
- lwjgl: Lightweight Java Game Library

### 4.1.1 Benchmark Execution Scripts

The execution of each benchmark is automated through a shell script as seen in Code 12. The fat-jar is executed once for each library and JMH then runs the benchmark code multiple times including warmup iterations.

```
1  BName="<benchmark-name>"
2  OutDir="$(date +"%Y-%m-%d_%H-%M-%S")"
3
4  java -Xmx8g -jar benchmark-runner.jar "lutils.$BName" "$OutDir"
5  java -Xmx8g -jar benchmark-runner.jar "ffma.$BName" "$OutDir"
6  java -Xmx8g -jar benchmark-runner.jar "jna.$BName" "$OutDir"
7  java -Xmx8g -jar benchmark-runner.jar "lwjgl.$BName" "$OutDir"
```

Code 12: Shell script to execute a specific benchmark.

### 4.1.2 Result Graph Generation

After the benchmarks have completed a python script is executed, which reads all result JSON-files and generates a boxplot diagram from the measured execution times and a bar graph from the measured allocation rates. A boxplot is used for the execution times, because they sometimes exhibit higher variability. The boxplot diagram shows a box which extends from the first quartile to the third quartile. The orange line within the box describes the median. Additionally, the whiskers extending from the box show the farthest data points lying within 1.5 times the inter-quartile range (IQR $= Q_3 - Q_1$). Because each whisker must represent an actual observed data value, their length may differ, even though the $1.5 \cdot \text{IQR}$ cutoff distance is the same for both sides. Outliers outside that range are marked as empty circles. The boxplots of benchmark 2 and 3 of each experiment feature a purple line, which shows the average of the reference benchmark.

A bar diagram is used for the allocation measurement results, because they exhibit minimal variability. Only JNA displays a mentionable variability, which is usually very low compared to the respective average values ($\text{std}/\text{avg} \leq 3\%$). However, in benchmark 1 of experiment 1 the measurements show extreme outliers in both allocation rate and execution time (This is

mentioned again in the result section of the affected benchmark). That is why the bar diagrams display the median instead of the average. For all other libraries the median and average values are almost identical due to the low variability. Furthermore benchmark 2 and 3 feature an additional purple bar, which shows the median of the reference benchmark.

### 4.1.3 Benchmark JMH Configuration

Benchmark 1 and 2 of each experiment require the same JMH configuration:

- Benchmark mode: Average Execution time.
- Warmup: 50 iterations. 1 seconds per iteration.
- Measurement: 400 iterations. 1 seconds per iteration.
- Forks: 3

Since all measured invocations require less than $10^8$ ns, 400 iterations provide at least 4,000 invocations of benchmark code. Additionally, the measurements are run three times (3 forks) on fully clear instantiations of new JVMs. These show that the medians vary less than 5% across all three forks. Except in benchmark 2 of experiment 3, where FFMA exhibits a median variation of 8.29%.

Benchmark 3 requires a different configuration:

- Benchmark mode: Single shot time.
- Warmup: 0 iterations
- Measurement: 2 iterations
- Forks: 1,000

With this configuration, both the benchmark and its reference counterpart are each executed 1,000 times. Since the goal of benchmark 3 is to isolate one-time initialisations and per-structure setup costs, a reference benchmark is required which allocates the same structures and performs the same write/read operations. Although benchmark 2 would fit that requirement, it cannot be used as reference because its execution time is measured in a stable state. Results from previous studies show that benchmark invocations during warmup can be almost six times slower than invocations once a stable state has been reached [24]. Furthermore, they show that this factor drops radically during the first 50 invocations. That is why a second iteration has been introduced to the configuration of benchmark 3, which allocates the same structures and executes the same write/read operations without performing one-time initialisations and per-structure setup, because those have already been completed in the previous iteration. Since the second iteration represents the second invocation of the benchmark code it is expected that a stable state has not been reached. However, it is possible that some JIT-optimisations have already been made, which is not avoidable.

## 4.2 Experiment 1

Experiment 1 evaluates small structures using the four different java native access libraries: LUtils, FFMA, JNA, LWJGL. The structures all have sizes between 8 and 48 bytes. Additionally, they contain up to three different element types. Even though each element can be another structure containing more element types, the size constraint retains a low amount of different element types. Notably, the structure with greatest complexity contains five different element types.

In this experiment, benchmark 1 allocates ten different structures without reading or writing to them. This provides variation in memory layouts, which helps to capture a broader range of real-world usage scenarios. Additionally, it reduces the risk of bias caused by library specific optimisations or inefficiencies. Benchmark 2 and 3 allocates two different structures to write to and read from. Only two structures are used, because each structure provides multiple read and write operations.

### 4.2.1 Benchmark 1 Results

The performance and memory allocation results of benchmark 1 are summarised in Fig. 3 and Tab. 1. They reveal large differences between the libraries in both execution time and Java-Heap allocation. Noteworthy is that the outliers have been hidden from the execution time boxplot diagram, because JNA displayed extreme variability, which would result in an unreadable plot. Furthermore, the y-axis has been truncated to allow all libraries to be displayed within the boxplot. Thus, the absolute difference between FFMA and LUtils is substantially larger than what the plot visually suggests.

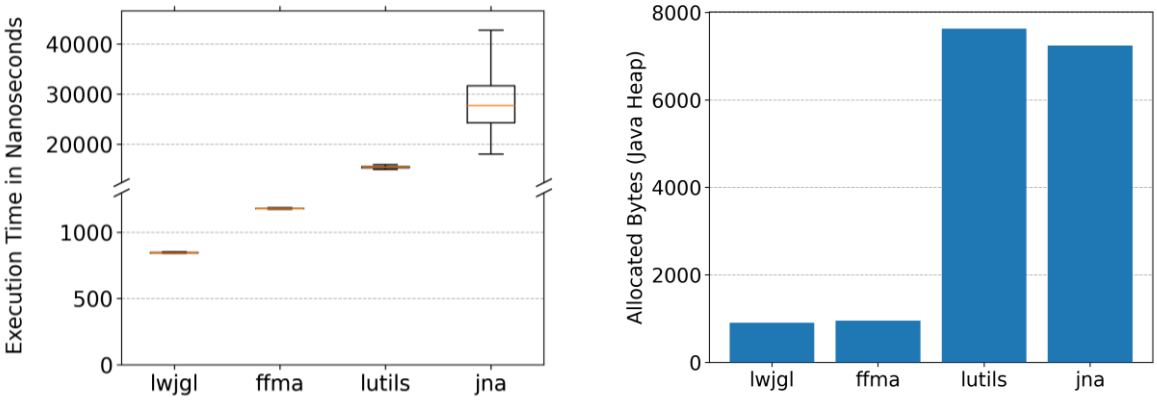


Fig. 3: Results for benchmark 1 of experiment 1. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Execution Time (ns) | Ratio to LWJGL | Median Allocated Bytes |
|---|---|---|---|
| LWJGL | 847 | 1.00 | 904 |
| FFMA | 1,180 | 1.39 | 952 |
| LUtils | 15,418 | 18.21 | 7,632 |
| JNA | Med= 27,703 38,868 | 45.91 | 7,240 |

Tab. 1: Results for benchmark 1 of experiment 1. Average execution time in nanoseconds and median allocated bytes of benchmark 1 of experiment 1. For JNA the median is presented as well, because its average is affected by outliers, which is further discussed below.

LWJGL and FFMA exhibit the best performance in terms of execution time. The lowest average execution time is achieved by LWJGL with 847 ns, followed by FFMA with 1180 ns. Both outperform the remaining libraries by more than an order of magnitude. LUtils and JNA show much higher execution times of 15,418 ns and 38,868 ns respectively. LWJGL and FFMA have a very low variability in execution time, with standard deviations less than 4 ns. LUtils exhibits a moderately higher standard deviation of 215 ns. In contrast, JNA demonstrates substantial variability, with a standard deviation of 48,582 ns, including 55 outliers up to values of 652,340 ns. That is why the median might be more representative for the library JNA which is 27,703 ns.

A similar trend can be observed for the Java-Heap memory allocation. LWJGL and FFMA allocate the smallest amount of heap memory, with medians of 904 bytes and 952 bytes respectively. In distinction, LUtils and JNA allocate approximately eight times more heap memory, with 7,632 bytes and 7,240 bytes at the median. As previously stated, the heap allocation

measurements exhibit small variability. LWJGL, FFMA and LUtils show negligible variation, with standard deviations below one byte. Only JNA displays a standard deviation of 10,609 bytes, due to outliers reaching up to 219,739 bytes. However, 99% of all JNA measurements are below 8,040 bytes.

Overall, the results indicate that LWJGL and FFMA provide the most efficient implementation when creating small structures among the evaluated libraries achieving both the lowest execution time and the lowest allocation rate on the Java-Heap. In contrast, LUtils and JNA introduce significant performance and memory overhead. Additionally, JNA exhibits very large outliers in execution time and memory allocation.

### 4.2.2 Benchmark 1 Interpretation

LWJGL's average execution time is slightly lower than the average execution time of FFMA. This is likely due to the different allocation strategies used by the libraries. LWJGL is allocating using the native malloc function from jemalloc, while FFMA uses its `Arena` to allocate memory, which provides more safety and correctness guarantees as well as zero-initialised memory.

JNA and LUtils exhibit execution times that are more than six times higher compared to the other evaluated libraries. This behaviour can primarily be attributed to their reliance on reflection and the complex class hierarchies required to generate structure representations dynamically at runtime. In addition, both libraries perform validation steps each time a structure instance is created, introducing further overhead. Furthermore, LUtils encapsulates primitive native types within dedicated wrapper classes. This design increases both execution time and allocation rate, as each primitive value is represented by an object rather than a Java primitive type. In contrast, JNA maps native primitive types directly to their corresponding Java primitive types. This is likely the reason why JNA requires slightly less Java-Heap memory when working with small structures consisting mostly of primitive types.

Another noteworthy observation is that JNA's large variation in execution time and the extreme outliers regarding allocation rate is unexpected and could be caused by some specific implementation only used by JNA.

### 4.2.3 Benchmark 2 Results

Fig. 4 and Tab. 2 summarise the performance and memory allocation results of benchmark 2. They reveal differences between the libraries in write and read operations regarding both execution time and allocation rate. As described in 4.1.2, in addition to the primary measurements, a reference benchmark is included, which allocates the same structures without executing any write/read operations.

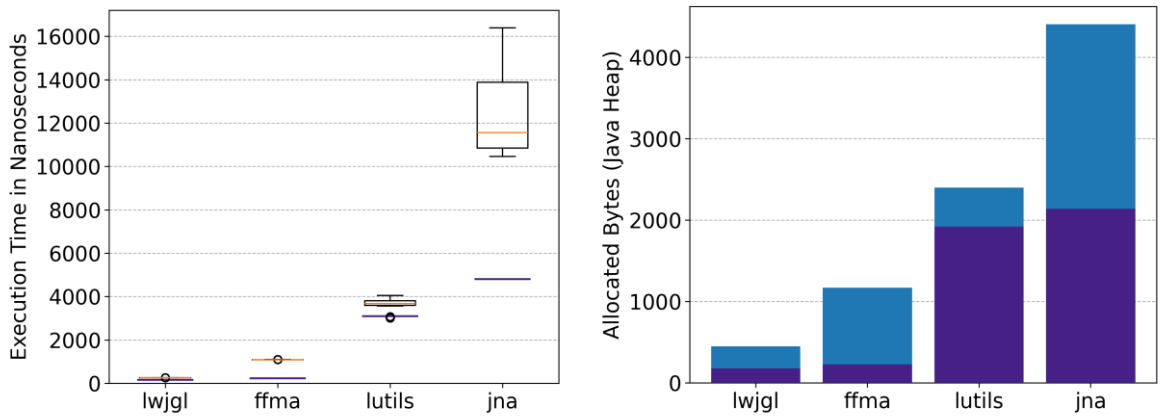


Fig. 4: Results for benchmark 2 of experiment 1. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Exec. Time (ns) | Ratio to LWJGL | Reference | M. Alloc. Bytes | Reference |
|---|---|---|---|---|---|
| LWJGL | 251 | 1.00 | 150 | 448 | 176 |
| FFMA | 1,084 | 4.31 | 228 | 1,168 | 224 |
| LUtils | 3,634 | 14.44 | 3,084 | 2,400 | 1,920 |
| JNA | 12,330 | 48.97 | 4,805 | 4,404 | 2,137 |

Tab. 2: Results for benchmark 2 of experiment 1. Average execution time in nanoseconds and median allocated bytes of benchmark 2 from experiment 1 as well as the respective reference benchmark.

Like the results of benchmark 1, LWJGL achieves the lowest average execution time at 251 ns, followed by FFMA with 1,084 ns. Once again outperforming LUtils and JNA, which report average execution times of 3,634 ns and 12,330 ns respectively. Noteworthy is that the gap between LWJGL and FFMA substantially increased compared to benchmark 1. The variations show a similar trend to benchmark 1. LWJGL and FFMA display a low variability in execution time, with standard deviations less than 3 ns. LUtils and JNA exhibit slightly higher standard deviations of 250 ns and 1,583 ns. Compared to benchmark 1 the standard deviations of JNA have sunken by a factor of 30. Furthermore, JNA does no longer exhibit outliers.

Comparison with the reference benchmark shows that LWJGL introduces the smallest and FFMA the largest relative write/read overhead for writing to and reading from the structures. The execution time of LWJGL increases from 150 ns to 251 ns (101 ns increase; 40% overhead), corresponding to the smallest absolute overhead. FFMA rises from 228 ns to 1,084 ns (855 ns increase; 79% overhead), which is the largest relative overhead. LUtils and JNA exhibit

comparatively smaller proportional increases relative to their already high reference values. LUtils increases from 3,084 ns to 3,634 ns (549 ns increase; 15% overhead), corresponding to the smallest relative overhead. JNA rises from 4,805 ns to 12,330 ns (7526 ns increase; 61% overhead), resulting in the largest absolute overhead.

The Heap allocation results show a similar pattern. LWJGL again demonstrates the lowest allocation footprint with a median of 448 bytes, followed by FFMA with 1,168 bytes. Thus, once again outperforming LUtils and JNA, which exhibit allocation measurements of 2,400 bytes and 4,404 bytes respectively. Analogous to the execution time results, the disparity between LWJGL and FFMA has substantially increased compared to benchmark 1. It increased from an almost equivalent allocation rate to a difference of a factor of 2.6. Furthermore, the difference between FFMA and LUtils shrinks from a factor of eight to a factor of two. Once again, the heap allocation measurements exhibit small variability across all libraries. FFMA, LWJGL and LUtils exhibit standard variations of less than one byte and JNA results in a standard deviation of 70 bytes. Unlike the previous benchmark the standard deviation from JNA is not caused by outliers.

Comparison with the reference benchmark shows that allocation increases across all libraries. LWJGL shows the smallest absolute increase rising from 176 bytes to 448 bytes (272 bytes increase; 61% overhead). In Contrast, JNA displays the largest absolute increase rising from 2,137 bytes to 4,404 bytes (2,267 bytes increase; 51% overhead). The smallest relative overhead exhibits LUtils, which rises from 1,920 bytes to 2,400 bytes (480 bytes increase; 20% overhead), On the other hand, FFMA displays the largest relative overhead, rising from 224 bytes to 1,168 bytes (944 bytes increase; 81% overhead)

Overall, the results indicate again that LWJGL provides the most efficient implementation for write and read access, both in terms of execution time and memory allocation rate. FFMA still outperforms LUtils and JNA, but displays the largest relative write and read overhead of approximately 80% in both execution time and memory allocation.

### 4.2.4 Benchmark 2 Interpretation

Comparing the results to the reference benchmark reveals some interesting observations. Firstly, the large relative write/read overhead of LWJGL is unexpected, because the generated LWJGL structure classes translate memory operations directly to `Unsafe.put*()` methods. Looking at the benchmark code, this can be reasonably explained. A large part of the introduced overhead stems from the benchmark code itself, specifically from `Integer`, `Long` and `Float` wrapper classes that are created during the benchmark. Of course, this overhead is introduced to all libraries equally, but the effect is most visible on LWJGL, due to its low reference execution time and allocating rate.

Another interesting observation is the large absolute overhead of FFMA introduced by writing and reading from the structures. This is likely due to the use of `VarHandle` instead of written or generated functions, which requires additional conversions and checks to be executed on every write and read operation[26].

LUtils has the smallest relative overhead, because similar to LWJGL it translates write and read operations directly to `ByteBuffer.put()` and `ByteBuffer.get()` methods. LWJGL remains faster, because unlike `Unsafe`, `ByteBuffer` executes some checks on every write and read operation. This is further discussed in 5.2.1 (paragraph Write and Read Operations).

JNA exhibits the largest absolute overhead. This is due to the way the JNA Structures are implemented. The structure fields are represented by normal Java types. Thus, when writing to a structure all values are written to the Java-Heap. Only after calling the `Structure.write()` method, they are read using Java reflection and then converted and written to native memory. Reading from the structure requires a similar pipeline.

Another observation is, that the large variations in execution time from JNA have sunken from benchmark 1 to benchmark 2. To provide a better comparison the standard deviation of each benchmark is compared proportional to the respective average execution time. For JNA $std/\text{avg}$ sinks from 125% to 13%. This suggests that outliers in benchmark 1 might be caused by allocating the native memory or initialising the structure classes, because this benchmark creates less structures.

[26] https://github.com/openjdk/jdk/blob/a35945ae067ffd60d5f3740600866506 36ebd9de/src/java.base/share/classes/java/lang/invoke/VarHandle.java#L474

### 4.2.5 Benchmark 3 Results

Fig. 5 and Tab. 3 summarise the performance and memory allocation results of benchmark 3, which evaluate the startup overhead regarding both execution time and allocation rate of the examined native access libraries. As described previously, this benchmark captures one-time initialisation costs and per-structure setup costs and does not allow a separation of these components. Furthermore, as described in 4.1.2, in addition to the primary measurements, a reference benchmark is included, which allocates the same structures and executes the same write/read operations after a single warmup invocation, to isolate the costs mentioned above. Due to the substantially smaller measurements of the reference benchmark, all reference lines are so close to the bottom of the plots, that they are almost not visible.

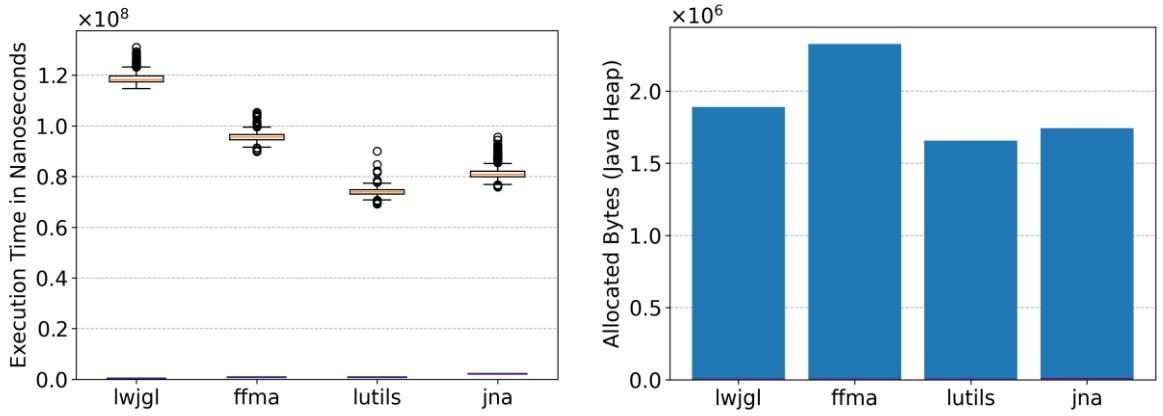


Fig. 5: Results for benchmark 3 of experiment 1. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Exec. Time (ns) | Ratio to LWJGL | Reference | M. Alloc. Bytes | Reference |
|---|---|---|---|---|---|
| LWJGL | 118,980,574 | 1.00 | 285,295 | 1,888,384 | 7,784 |
| FFMA | 95,631,635 | 0.80 | 787,491 | 2,325,744 | 7,952 |
| LUtils | 74,026,697 | 0.62 | 860,963 | 1,657,048 | 9,176 |
| JNA | 81,609,535 | 0.69 | 2,200,692 | 1,741,968 | 11,864 |

Tab. 3: Results for benchmark 3 of experiment 1. Average execution time in nanoseconds and median (M.) allocated bytes of benchmark 3 from experiment 1 as well as the respective reference benchmark.

In terms of execution time, LUtils achieves the lowest average at $74 \cdot 10^6$ ns, followed by JNA and FFMA with $82 \cdot 10^6$ ns and $96 \cdot 10^6$ ns respectively. LWJGL exhibits the highest execution time at $119 \cdot 10^6$ ns. All libraries exhibit relatively large standard deviations. JNA shows the largest dispersion, with a standard deviation of $3.0 \cdot 10^6$ ns, while LUtils exhibits the lowest variability at $1.4 \cdot 10^6$ ns.

Java-Heap allocation follows a similar pattern. Once again, LUtils demonstrates the lowest median value at $1.66 \cdot 10^6$ bytes, again followed by JNA with $1.74 \cdot 10^6$ bytes. LWJGL exhibits an allocation rate of $1.89 \cdot 10^6$ bytes. Unlike the execution times, FFMA displays the highest allocation rate, allocating $2.33 \cdot 10^6$ bytes. Consistent with the previous benchmarks, Java-Heap allocation variability remains low across all evaluated libraries, with FFMA exhibiting the highest standard deviation at 99 bytes.

When comparing to the reference benchmark, the execution times are between 37 and 417 times higher for all libraries. The smallest relative increase is exhibited by JNA due to its already high reference value. The Java-Heap memory allocation results are more than 146 times higher across all libraries.

Overall, this benchmark shows, that the initialisation costs across all libraries are much higher than the costs for creating, writing to and reading from two different structures.

### 4.2.6 Benchmark 3 Interpretation

The results demonstrate that all libraries require considerable startup costs. Furthermore, it is likely that most of the execution time and memory allocation is required for one-time initialisations like class loading, native library initialisations and one-time runtime configurations. However, this interpretation must be treated with caution, as it cannot be confirmed by this benchmark.

One notable observation is that the libraries LUtils and JNA, which perform worse relative to LWJGL and FFMA in benchmark 1 and 2, achieve better results in this benchmark. This suggests, that LWJGL and FFMA might perform more one-time initialisations and optimisation steps during application startup and structure related class loading, which leads to additional upfront overhead but subsequently enables better performance during repeated structure usage.

## 4.3 Experiment 2

Experiment 2 evaluates medium structures across the four libraries. The structures contain between five and ten different element types and exhibit sizes between 112 bytes and 424 bytes. Once again, each element can be another structure which may contain more element types. Thus, comparing structure creation, write/read overhead and startup costs with more complex structures.

Benchmark 1 allocates five different structures without reading or writing to them. Once again, this provides variation in memory layout and reduces the risk of bias caused by library specific optimisations or inefficiencies. Benchmark 2 and 3 allocate a single structure that is subsequently used for write and read operations. Although only one structure is created, the benchmark still introduces variability due to the structure's complexity and its large number of fields, which are all used for write/read operations.

### 4.3.1 Benchmark 1 Results

The performance results of benchmark 1 of experiment 2 are presented in Fig. 6 and Tab. 4. To allow all libraries to be displayed within the figures, the y-axis of both the execution time and the allocation rate plot contain a broken axis. Thus, the absolute difference between FFMA and LUtils is substantially larger than what the plot visually suggests.

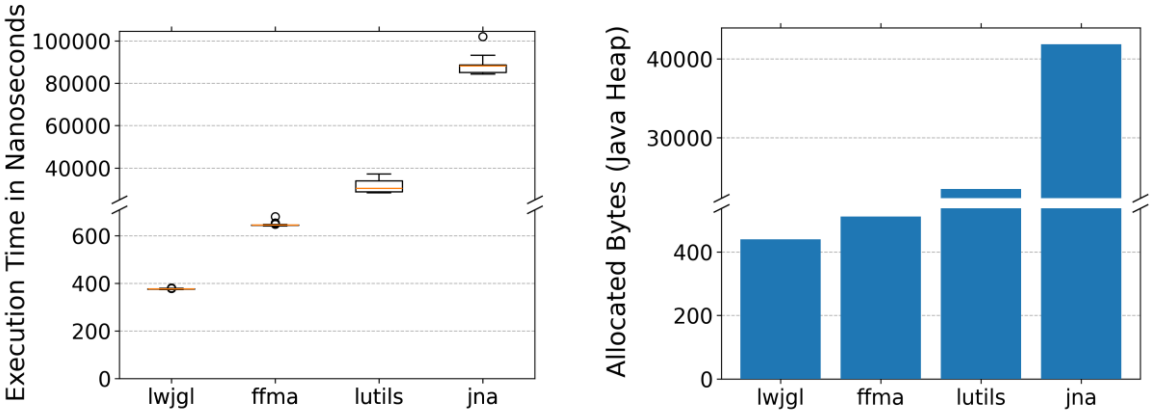


Fig. 6: Results for benchmark 1 of experiment 2. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Execution Time (ns) | Ratio to LWJGL | Median Allocated Bytes |
|---|---|---|---|
| LWJGL | 376 | 1.00 | 440 |
| FFMA | 644 | 1.71 | 512 |
| LUtils | 31,322 | 83.37 | 23,472 |
| JNA | 87,122 | 231.90 | 41,856 |

Tab. 4: Results for benchmark 1 of experiment 2. Average execution time in nanoseconds and average allocated bytes of benchmark 1 from experiment 2.

The results show a similar pattern to benchmark 1 of experiment 1. In terms of execution time, LWJGL achieves the best performance with an average execution time of 376 ns, followed by FFMA at 644 ns. Both libraries outperform LUtils and JNA by more than one order of magnitude. They report average executions times of 31,322 ns and 87,122 ns respectively. LWJGL and FFMA exhibit negligible variability with standard deviations less than 3 ns. In contrast, JNA and LUtils demonstrate higher variability with standard deviations of 1975 ns and 2631 ns respectively.

The Java-Heap memory allocation results display a similar trend. LWJGL and FFMA show the best performance with medians of 440 bytes and 512 bytes. In contrast, LUtils and JNA allocate significantly more heap memory at 23,472 bytes and 41,856 bytes respectively. Once again, the variability across all four libraries is negligible. LWJGL, FFMA and LUtils exhibit standard deviations close to 0 bytes (< 0.03 bytes). JNA reports a standard deviation of approximately 90 bytes, which is less than 0.22% relative to the median.

Compared to benchmark 1 of experiment 1, the gap between the faster and slower libraries has increased substantially. For example, while LUtils is approximately 18 times slower than LWJGL in experiment 1, this disparity widens further in the current experiment, where LUtils exhibits an execution time approximately 83 times higher than that of LWJGL. A similar trend can be observed for the Java-Heap allocation measurements. Another noteworthy observation is that in experiment 1 LUtils and JNA had similar medians regarding allocation rate, but in this benchmark JNA allocates almost twice as much. Furthermore, JNA only contains a single execution time outlier and std/avg sinks from 124% in the previous experiment to 2.27% in this experiment.

Overall, the results indicate again that LWJGL and FFMA provide the most efficient implementation when creating and allocating complex structures among the evaluated libraries. Both achieve the lowest average execution time and the lowest average allocation rate. In contrast, LUtils and JNA introduce more than one order of magnitude longer execution times and Java-Heap memory allocations than LWJGL and FFMA.

### 4.3.2 Benchmark 1 Interpretation

Once again, the very low execution times of LWJGL and FFMA are expected, because both libraries do not require any reflective access when the structure is created. What is noteworthy is that FFMA is still slower than LWJGL. This is interesting because FFMA only allocates the memory without having to initialise any custom classes for each structure. Meanwhile LWJGL initialises the generated classes, which enable simple read and write access to the structures. This disparity is likely once again due to the different allocators employed by the libraries and the fact, that LWJGL does not initialise the allocated memory with zeros. A similar observation can be made about the Java-Heap allocation rates of LWJGL and FFMA. Again, FFMA allocates more bytes on the Java-Heap than LWJGL. This suggests, that the allocator FFMA uses requires more Java-Heap memory than LWJGL's allocator.

Like benchmark 1 of experiment 1 JNA and LUtils require more than one order of magnitude more execution time and allocation rate than LWJGL. Once again, this is not unexpected due to their use of reflection and more complex classes required to dynamically create the structures at runtime. An interesting observation is that unlike experiment 1 JNA now exhibits an almost two times higher allocation rate than LUtils. This suggests that JNA might perform more complex runtime validations and initialisations.

### 4.3.3 Benchmark 2 Results

The results of benchmark 2 are displayed in Fig. 7 and Tab. 5. They allow execution time and Java-Heap memory allocation comparisons of write and read operations across the four libraries. As previously stated in 4.1.2, in addition to the primary measurements, a reference benchmark, allocating identical structures without performing any write/read operations, is included.

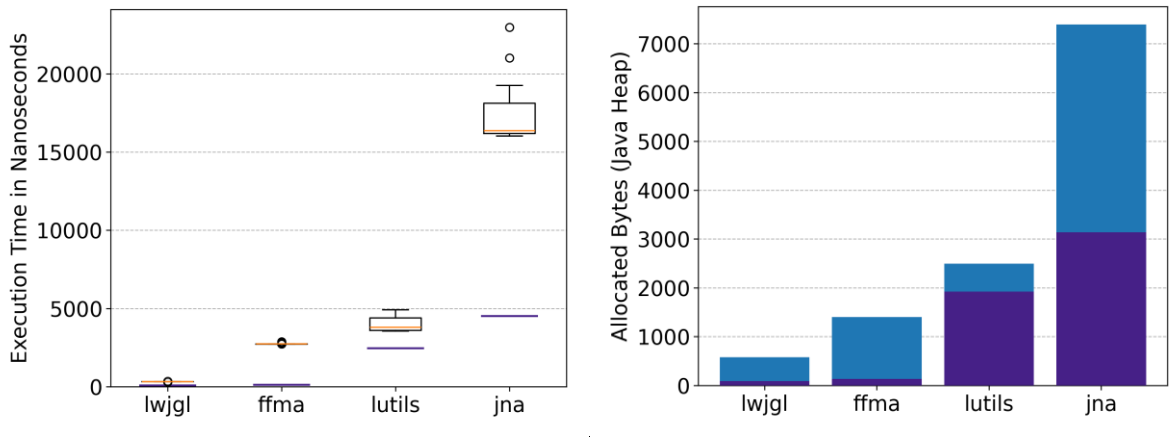


Fig. 7: Results for benchmark 2 of experiment 2. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Exec. Time (ns) | Ratio to LWJGL | Reference | M. Alloc. Bytes | Reference |
|---|---|---|---|---|---|
| LWJGL | 318 | 1.00 | 73 | 576 | 88 |
| FFMA | 2,730 | 8.59 | 124 | 1,400 | 136 |
| LUtils | 3,990 | 12.55 | 2,446 | 2,496 | 1,920 |
| JNA | 16,944 | 53.31 | 4,505 | 7,392 | 3,136 |

Tab. 5: Results for benchmark 2 of experiment 2. Average execution time in nanoseconds and median allocated bytes of benchmark 2 from experiment 2 and the respective reference benchmark.

With respect to execution time, LWJGL achieves the best performance with an average of 318 ns, followed by FFMA and LUtils exhibiting an approximately one order of magnitude higher average execution times at 2,730 ns and 3,990 ns respectively. JNA exhibits an even higher execution time at 16,944 ns. LWJGL and FFMA exhibit the lowest absolute standard deviations with approximately 1 ns and 12 ns respectively. LUtils and JNA demonstrate higher absolute standard deviations at 406 ns and 1006 ns.

Comparing with the reference benchmark shows that LWJGL introduces the smallest absolute overhead associated with write and read operations. Its execution time increases from 73 ns on

average to 318 ns (244 ns increase; 77% overhead). FFMA introduces a substantially larger absolute overhead and the largest relative overhead, with execution time rising from 124 ns to 2,730 ns (2,606 ns increase; 95% overhead). LUtils and JNA show smaller proportional increases compared to their already high reference values. LUtils exhibits the smallest relative overhead, rising from 2,446 ns in the reference benchmark to 3,990 ns (1,543 ns increase; 39% overhead). JNA exhibits by far the largest absolute increase with 4,505 ns in the reference case to 16,944 ns in the full benchmark (12,438 ns increase; 73% overhead).

A similar pattern is observed for heap allocation behaviour. LWJGL displays the lowest allocation rate with a median of 576 bytes, followed by FFMA and LUtils with 1,400 bytes and 2,496 bytes respectively. The highest number of allocated bytes is again displayed by JNA with a median of 7,392 bytes. Like the previous benchmarks, the variability across all four libraries is negligible. LWJGL, FFMA and LUtils exhibit standard deviations close to zero bytes (< 0.005 bytes). JNA reports a standard deviation of approximately 42 bytes, which is less than 0.6% proportional to the median.

Relative to the reference benchmark LWJGL displays the smallest absolute increase, rising from 88 bytes to 576 bytes (488 bytes increase; 85% overhead), followed by LUtils with the smallest relative overhead, increasing from 1,920 bytes to 2,496 bytes (576 bytes increase; 23% overhead). FFMA exhibits the largest relative write/read overhead, rising from 136 bytes to 1,400 bytes (1,264 bytes increase; 90% overhead). The library with the largest absolute increase is JNA, which rises from 3,136 bytes to 7,392 bytes (4,256 bytes increase; 58% overhead).

Compared to benchmark 2 of experiment 1, the results of this benchmark show a similar trend. The gaps between the different libraries have not drastically changed. Furthermore, while the absolute overhead in both execution time and allocated bytes has increased across all libraries, the relative overhead displays only minimal changes.

Overall, the results indicate that LWJGL again provides the most efficient implementation for write and read access, both in terms of execution time and memory allocation overhead. FFMA still outperforms LUtils and JNA, but displays the largest relative write and read overhead of approximately 90% in both execution time and memory allocation.

### 4.3.4 Benchmark 2 Interpretation

The results of benchmark 2 of this experiment are very similar to the results of benchmark 2 from experiment 1. Hence, these results can be interpreted similarly. That is why only the main points are listed shortly, and the interpretation can be assumed analogue to 4.2.4:

- The large relative write/read overhead of LWJGL is unexpected but can be reasonably explained due to the benchmark code.
- The large relative overhead of FFMA can likely be explained by the use of `VarHandle` instead of written or generated functions.
- LUtils has the smallest relative overhead, because it translates write and read directly to `ByteBuffer.put()` and `ByteBuffer.get()` methods.
- JNA exhibits the largest absolute overhead, this suggests, that the conversion of the values from the Java-Heap and then writing the converted values to the native memory as well as the reverse process for reading is very expensive.

Only the last observation from 4.2.4 - the large reduction in execution time variability for JNA from benchmark 1 to benchmark 2 - cannot be confirmed in this experiment. Specifically, std/avg rises from 2.27% to 5.93%. This weakens the earlier suggestion that the large variations may stem from allocating the native memory or initialising the structure classes.

### 4.3.5 Benchmark 3 Results

Fig. 8 and Tab. 6 summarise the performance and memory allocation results of benchmark 3. This benchmark evaluates the startup overhead without warmup iterations. As mentioned before, this benchmark captures one-time initialisation costs as well as per-structure setup costs and does not allow a separation of these components. Therefore, the results must be interpreted with care. Furthermore, as discussed in 4.1.2, in addition to the primary measurements, a reference benchmark is included, which allocates the same structures and executes the same read or write operations with a single warmup invocation, to isolate the costs mentioned above. Due to the substantially smaller measurements of the reference benchmark, all reference lines are so close to the bottom of the plots, that they are almost not visible.

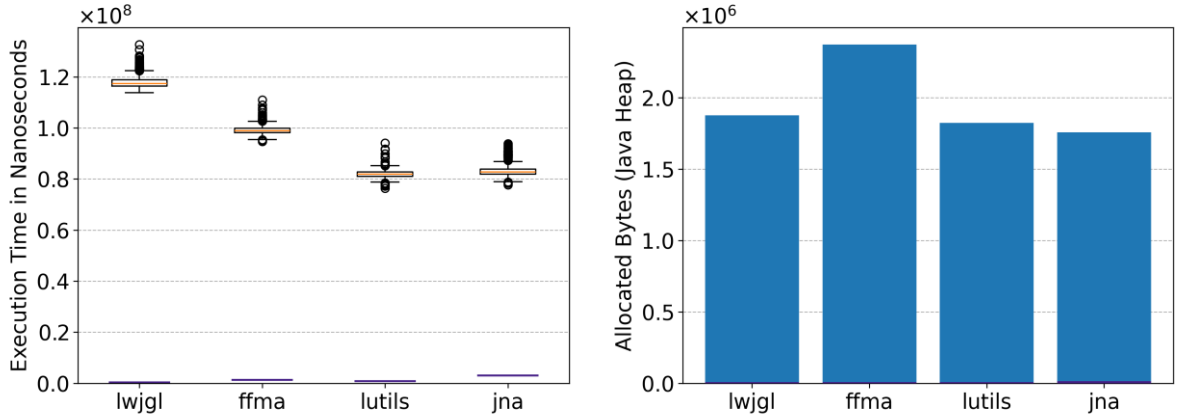


Fig. 8: Results for benchmark 3 of experiment 2. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Exec. Time (ns) | Ratio to LWJGL | Reference | M. Alloc. Bytes | Reference |
|---|---|---|---|---|---|
| LWJGL | 117,929,805 | 1.00 | 295,185 | 1,876,496 | 7,864 |
| FFMA | 99,102,727 | 0.84 | 1,320,737 | 2,371,496 | 8,272 |
| LUtils | 81,917,281 | 0.69 | 825,172 | 1,823,784 | 9,256 |
| JNA | 83,351,354 | 0.71 | 3,099,412 | 1,756,368 | 15,144 |

Tab. 6: Results for benchmark 3 of experiment 2. Average execution time in nanoseconds and median (M.) allocated bytes of benchmark 3 from experiment 2 as well as the respective reference benchmark.

With respect to execution time, LUtils achieves the lowest average startup time at $82 \cdot 10^6$ ns, followed closely by JNA with $83 \cdot 10^6$ ns. FFMA and LWJGL display noticeably higher execution times with an average of $99 \cdot 10^6$ ns and $118 \cdot 10^6$ ns respectively. All libraries exhibit relatively large standard deviations. JNA shows the largest dispersion, with a standard deviation of $2.6 \cdot 10^6$ ns, while LUtils exhibits the lowest variability at $1.5 \cdot 10^6$ ns.

The Java-Heap allocations show a slightly different trend. JNA and LUtils demonstrate the lowest median allocation rate at $1.76 \cdot 10^6$ ns and $1.82 \cdot 10^6$ ns respectively, closely followed by LWJGL with $1.88 \cdot 10^6$ ns. FFMA exhibits the highest allocation rate with $2.37 \cdot 10^6$ ns. Once again, Java-Heap allocation variability remains low across all evaluated libraries proportional to their high median values. FFMA exhibits the highest standard deviation at approximately 284 bytes.

Compared to the reference benchmark, the execution times are again between 26 and 400 times higher for all libraries. The smallest relative increase is exhibited by JNA due to its already high reference value and the largest relative increase is displayed by LWJGL. The Java-Heap memory allocation results are more than 115 times higher across all libraries.

When comparing this benchmark to benchmark 3 of experiment 1, the execution times across all libraries except LWJGL increased while the allocation rate remained almost constant. The largest increase in execution time displayed LUtils with an absolute increase of $7.9 \cdot 10^6$ ns, which is a relative increase of about 11% proportional to the results of experiment 1. In Contrast LWJGL decreased its execution time by $1.05 \cdot 10^6$ ns, which is a relative decrease of only 0.88% proportional to the results of experiment 1. In terms of allocation rate, only LUtils showed a noticeable increase, rising by approximately $166{,}736$ bytes which is a relative increase of 10% proportional to the results of experiment 1.

Overall, this benchmark shows like benchmark 3 of experiment 1, that the initialisation costs across all libraries are much higher than the costs for creating, writing to and reading from structures.

### 4.3.6 Benchmark 3 Interpretation

Although the structures in this benchmark are more complex, the observed execution times increases only marginally. This suggests that the additional per-structure initialisation cost is relatively small compared to the dominant one-time initialisation overhead, thereby supporting the interpretation made in benchmark 3 of experiment 1. This interpretation is expected to be further validated or challenged by benchmark 3 of the subsequent experiment, as it employs a much more complex structure.

Another noteworthy observation is that only LUtils displayed a large increase in both execution time and Java-Heap memory allocation of approximately 10% compared to benchmark 3 in experiment 1. This suggests that LUtils requires more initialisation logic for the complex structure compared to the other libraries.

## 4.4 Experiment 3

Experiment 3 evaluates large structures across all four libraries. The structures size of these structures is between 496,000 bytes and 1,184,000 bytes. The structures mostly contain arrays of primitive types or arrays of small or complex structures. Thus, this experiment's goal is to compare structure creation, writing and reading of very large and complex structures with a lot of arrays.

Once again, benchmark 1 allocates five different structures without reading or writing to them. Five structures are used to provide variation in memory layout and reduce the risk of bias caused by library specific optimisations or inefficiencies. Benchmark 2 and 3 allocate a single structure to write to and read from. The structure used is complex and provides a large variety of write and read operations.

### 4.4.1 Benchmark 1 Results

Fig. 9 and Tab. 7 present the results of benchmark 1 from experiment 3, which creates and allocates very large and complex structures without performing any write/read operations. Due to the large disparity between the evaluated libraries, the y-axis of both plots is once again broken, to improve visual readability. It should be noted that the relative difference between JNA and the other libraries are substantially larger than what is visually suggested.

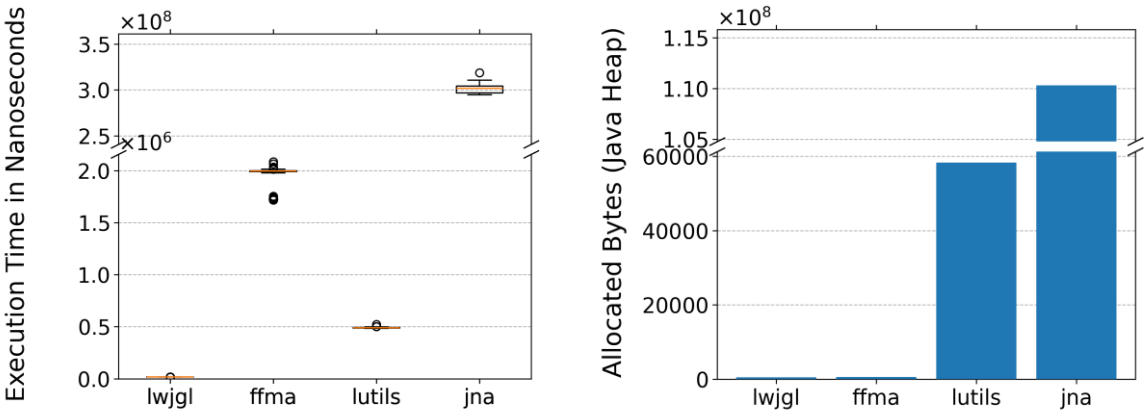


Fig. 9: Results for benchmark 1 of experiment 3. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Execution Time (ns) | Ratio to LWJGL | Ø Allocated Bytes |
|---|---|---|---|
| LWJGL | 16,513 | 1.00 | 440 |
| FFMA | 1,960,479 | 118.72 | 525 |
| LUtils | 489,899 | 29.67 | 58,363 |
| JNA | 301,014,709 | 18,228.94 | 110,294,524 |

Tab. 7: Results for benchmark 3 of experiment 3. Average execution time in nanoseconds and median allocated bytes of benchmark 1 from experiment 3.

Once again LWJGL outperforms all other libraries, achieving an average execution time of 16,513 ns. FFMA and LUtils exhibit more than one order of magnitude higher execution times with 1,960,479 ns and 489,899 ns respectively. This means, for the first time LUtils outperforms FFMA in terms of execution time. JNA demonstrates by far the worst performance with an average execution time of $301 \cdot 10^6$ ns. Thereby requiring 154 times more execution time than the second slowest library FFMA. All libraries report a low variability proportional to their average execution time with $\text{std}/\text{avg} \leq 5\%$ . LWJGL exhibits the smallest variability with only 81 ns. FFMA and LUtils display higher absolute variability at 92,392 ns and 3,015 ns respectively. JNA demonstrates the highest variability with $4.25 \cdot 10^6$ ns.

The Java-Heap allocation results show a slightly different trend. LWJGL and FFMA allocate minimal heap memory, with medians of 440 bytes and 525 bytes respectively. In contrast, LUtils and especially JNA exhibit substantially higher memory allocations at 58,363 bytes and $110 \cdot 10^6$ bytes respectively. Once again LWJGL, FFMA and LUtils exhibit minimal variability with standard deviations less than 2 bytes. In Contrast, JNA demonstrates a higher absolute standard deviation at 146,937 bytes. However, proportional to the high allocation rate median, this corresponds to a relative variation of less than 0.2%.

Compared to benchmark 1 of experiment 2, FFMA is now for the first time slower than LUtils. Furthermore, the performance gap between LWJGL and FFMA has increased substantially from an approximate factor of 1.71 to a factor of 118.72. On the other hand, the Java Heap allocation results remain similar to benchmark 1 of experiment 2. However, the gap between JNA and the second worst library has increased substantially for both execution time and allocation rate. In benchmark 1 of experiment 2, JNA requires 2.8 times more execution and 1.8 times more allocated bytes. In Contrast, in this experiment it requires 153 times more execution time and 1,890 times more allocated bytes than the second worst library.

Overall, the results indicate that LWJGL provides the most efficient implementation when creating and allocating large complex structures. In contrast FFMA, LUtils and especially JNA struggle with the allocation and creation of these structures. They require at least 30 times more execution time than LWJGL.

### 4.4.2 Benchmark 1 Interpretation

The very low execution time of LWJGL is again expected for multiple reasons. Firstly, it does not require any reflective access when the structure is created, and secondly it does not fill the allocated memory with zeros. However, noteworthy is the fact that LWJGL requires less allocated bytes than FFMA. This is interesting because LWJGL creates classes to easily perform

write and read operations on the structures while FFMA does not. These classes should require a large number of allocated bytes due to the high complexity of the nested structures. This disparity can be explained by the fact that the large structures mostly consist of arrays of other structures and arrays of primitive types. LWJGL does not initially create the structure classes for each array element. Instead, the classes are only created when required. This means that a large increase in allocated bytes is expected when write and read operations are performed in the next benchmark.

The fact that JNA requires 1,890 times more Java-Heap memory allocation than LUtils might be explained by two reasons. The first reason is that JNA is the only library, that stores the content of each structure on the Java-Heap before writing it to native memory. This might sound contradicting to the claim mentioned in the interpretation of benchmark 1 from experiment 1 (4.2.2), that LUtils' wrapper classes for primitive types result in more required Java-Heap space than the actual Java primitive type itself, which JNA uses. However, in this benchmark arrays of primitive types are used repeatedly. LUtils provides wrapper classes for these arrays, which require a similar amount of Java-Heap memory as the structure wrapper classes for primitive types. In Contrast, JNA uses Java primitive type arrays to represent native primitive type arrays. These require more bytes on the Java-Heap based on the length of the array. Thus, an array of 100,000 integers requires 400,000 bytes while LUtils requires less than 416 bytes (measured with JMH; contains overhead required for allocating memory). The second reason is that while LUtils does create a Java-Array for each structure array, like LWJGL it only initialises the elements of the array with structure specific classes when they are required for writing or reading. JNA on the other hand initialises every array element when the parent structure is created.

### 4.4.3 Benchmark 2 Results

The execution time and memory allocation results of benchmark 2 are presented in Fig. 10 and Tab. 8. Once again, an additional reference benchmark is included, which allocates the same structure without performing any write or read operations.

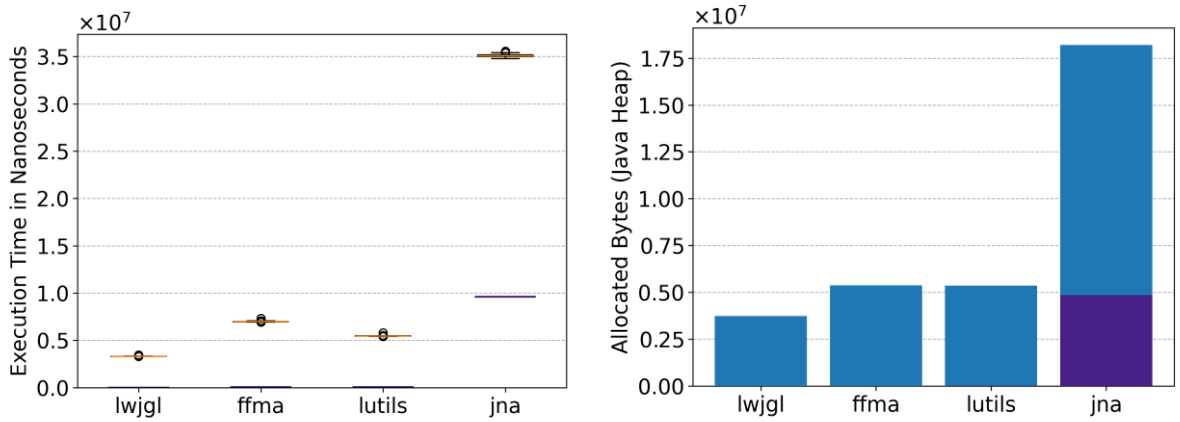


Fig. 10: Results for benchmark 2 of experiment 3. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Exec. Time (ns) | Ratio to LWJGL | Reference | Ø Allocated Bytes | Reference |
|---|---|---|---|---|---|
| LWJGL | 3,304,874 | 1.00 | 158 | 3,728,039 | 88 |
| FFMA | 6,996,741 | 2.12 | 49,061 | 5,369,712 | 136 |
| LUtils | 5,454,467 | 1.65 | 67,130 | 5,348,526 | 7,912 |
| JNA | 35,086,296 | 10.62 | 9,609,907 | 18,205,002 | 4,843,384 |

Tab. 8: Results for benchmark 2 of experiment 3. Average execution time in nanoseconds and median allocated bytes of benchmark 2 from experiment 2 as well as the respective reference benchmark.

In terms of execution time, LWJGL is once again the fastest library with an average of $3.3 \cdot 10^6$ ns, followed by LUtils and FFMA at $5.4 \cdot 10^6$ ns and $6.9 \cdot 10^6$ ns respectively. JNA demonstrates a substantially higher execution time than the other libraries, with an average of $35.1 \cdot 10^6$ ns, which is approximately one order of magnitude more execution time than the best performing library LWJGL. The absolute execution time variability across all four libraries is higher compared to previous benchmarks. LWJGL exhibits the smallest standard deviation with 12,842 ns, followed by LUtils and FFMA with an absolute variability of 23,240 ns and 33,845 ns respectively. The highest absolute standard deviation is displayed by JNA with 133,688 ns. When expressed as a proportion of the respective average execution times, all standard deviations are below 0.5%.

Comparing the results to the reference benchmark shows that all libraries exhibit a large write/read overhead. LWJGL, LUtils and FFMA increase their execution time by $3.3 \cdot 10^6$ ns, $5.4 \cdot 10^6$ ns and $6.9 \cdot 10^6$ ns respectively, which is a relative overhead of more than 98% for all three libraries. JNA displays a substantially higher increase inexecution with an absolute increase of $25.5 \cdot 10^6$ ns, which is, due to its already very high reference value, the smallest write/read overhead at 73%.

A similar pattern is displayed for the Java-Heap memory allocations. Once again LWJGL demonstrates the lowest median allocation rate at $3.7 \cdot 10^6$ bytes, followed by LUtils and FFMA with approximately $5.3 \cdot 10^6$ and $5.4 \cdot 10^6$ bytes. In contrast, JNA exhibits a substantially higher allocation rate at $18.2 \cdot 10^6$ bytes. Once again, the variability across all evaluated libraries is negligible. JNA exhibits the highest absolute variability, with a standard deviation of approximately 37 bytes.

Comparing the allocation rate to the reference benchmark shows that all libraries again demonstrate a large write/read overhead. For LWJGL, LUtils and FFMA this overhead exceeds 99%. Only JNA displays a smaller relative overhead with 73% due to its already high reference value. Nevertheless, JNA displays the largest absolute increase with $13.4 \cdot 10^6$ bytes.

Comparing these results to benchmark 2 of experiment 2 shows that the relative gaps between LWJGL, FFMA and LUtils have generally decreased. In Experiment 2 LWJGL is about nine times faster than the second fastest library, which has now decreased to a factor of approximately 1.7. Noteworthy is also that unlike before LUtils is now slightly faster than FFMA. Additionally, the absolute and relative write/read overhead in both execution time and allocated bytes has increased for all libraries. Especially for LUtils the relative execution time overhead has increased from 39% to more than 98%.

Overall, the results confirm again that LWJGL provides the most efficient implementation even when writing to a very large and complex structure, but the performance gap to the other libraries is smaller compared to earlier write/read benchmarks. JNA once again performs the worst out of all four evaluated libraries.

### 4.4.4 Benchmark 2 Interpretation

In this benchmark the large write and read overhead of LWJGL is not as unexpected as in the earlier write/read benchmarks. The reasoning for that is that LWJGL creates the custom structure classes for each element of all arrays whose component type is not a primitive type. These classes are required to write and read from the elements of the arrays. In fact, this reason also applies to FFMA and LUtils as well. FFMA creates a `MemorySegment` for each element and

LUtils creates its custom classes extending `ComplexStructure`. Only JNA initialises all array elements with its custom structure classes in advance while creating the parent structure, as mentioned in the interpretation of benchmark 1.

Furthermore, the following interpretations made in benchmark 2 of experiment 1 and 2 still hold:

- A part of the large relative write/read overhead of LWJGL is due to the benchmark code itself, specifically primitive type wrapper classes. This affects the absolute overhead equally for all libraries.
- A part of the large relative overhead of FFMA can likely be explained by the use of `VarHandle` instead of written or generated functions for write/read operations.
- LUtils translates write and read operations directly to `ByteBuffer.put()` and `ByteBuffer.get()` methods, which might be the reason why LUtils is now slightly faster than FFMA.

### 4.4.5 Benchmark 3 Results

The performance results of benchmark 3 are displayed in Fig. 11 and Tab. 9. Once again, this benchmark evaluates the startup overhead when allocating and writing to a single large and complex structure without warmup. As previously stated, an additional reference measurement is in included, which allocates the same structure and performs the same write/read operations with a single warmup invocation. Thereby isolating one-time initialisation costs and per-structure setup costs.

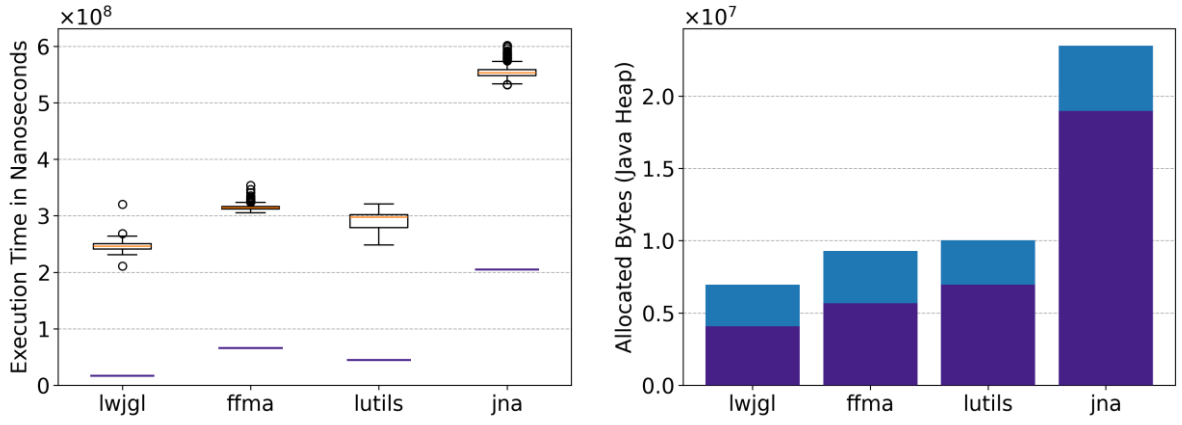


Fig. 11: Results for benchmark 3 of experiment 3. Left: Boxplot diagram of the execution times across all four libraries. Right: Bar chart of the median allocation rate across all four libraries.

| Library | Ø Exec. Time (ns) | Ratio to LWJGL | Reference | M. Alloc. Bytes | Reference |
|---|---|---|---|---|---|
| LWJGL | 246,183,961 | 1.00 | 16,792,717 | 6,951,984 | 4,070,760 |
| FFMA | 314,586,644 | 1.28 | 65,757,495 | 9,289,368 | 5,669,280 |
| LUtils | 292,255,618 | 1.19 | 44,551,316 | 10,036,336 | 6,955,592 |
| JNA | 554,525,448 | 2.25 | 204,618,086 | 23,492,040 | 18,973,680 |

Tab. 9: Results for benchmark 3 of experiment 3. Average execution time in nanoseconds and median (M.) allocated bytes of benchmark 3 from experiment 3 as well as the respective reference benchmark.

In terms of execution time LWJGL achieves the lowest average startup time with $246 \cdot 10^6$ ns, followed by LUtils and FFMA at $292 \cdot 10^6$ ns and $314 \cdot 10^6$ ns respectively. JNA exhibits a noticeable higher execution time with an average of $554 \cdot 10^6$ ns. The variability is noticeable across all four libraries. FFMA shows the smallest absolute variability with a standard deviation of $4.53 \cdot 10^6$ ns. In Contrast, LUtils exhibits the highest standard deviation at $12.55 \cdot 10^6$ ns.

In terms of Java-Heap memory allocation rate, LWJGL again achieves the lowest median of $6.9 \cdot 10^6$ bytes, followed by FFMA and LUtils with $9.3 \cdot 10^6$ bytes and $10.0 \cdot 10^6$ bytes respectively. JNA demonstrates a noticeably higher median allocation rate at $23.5 \cdot 10^6$ bytes. The allocation variability of FFMA, LWJGL and LUtils is neglectable with standard deviations less than 707 bytes. Only JNA displays a substantially higher standard deviation at 49,001 bytes, which is still less than 1% proportional to its average value.

Compared to the reference benchmark, LWJGL, LUtils and FFMA increase their execution time by $229.4 \cdot 10^6$ ns, $247.7 \cdot 10^6$ ns and $248.8 \cdot 10^6$ ns respectively, which are relative overheads ranging from 79% to 93%. JNA displays a substantially higher absolute increase in execution time with $349.9 \cdot 10^6$ ns, which is, due to its already very high reference value, the smallest write/read overhead at 63%. A similar trend can be observed for the Java-Heap memory allocation rates. LWJGL exhibits the smallest absolute increase of $2.9 \cdot 10^6$ bytes (41% overhead), followed by LUtils and FFMA with $3.0 \cdot 10^6$ bytes (30% overhead) and $3.6 \cdot 10^6$ bytes (39% overhead) respectively. JNA exhibits the largest absolute, but smallest relative increase, rising by $4.5 \cdot 10^6$ bytes (19% overhead).

The comparison to benchmark 3 of the previous experiments, show that the execution times and allocation rate increased across all evaluated libraries. To isolate the startup overhead the corresponding reference values have been subtracted from the execution times of benchmark 3

of each experiment, resulting in the data presented in Fig. 12. Furthermore, the increase of these values from experiment 2 to experiment 3 – and from experiment 1 to experiment 3 – have been calculated, as well as the factor by which these values have risen. The results of the calculations are presented in Tab. 10.

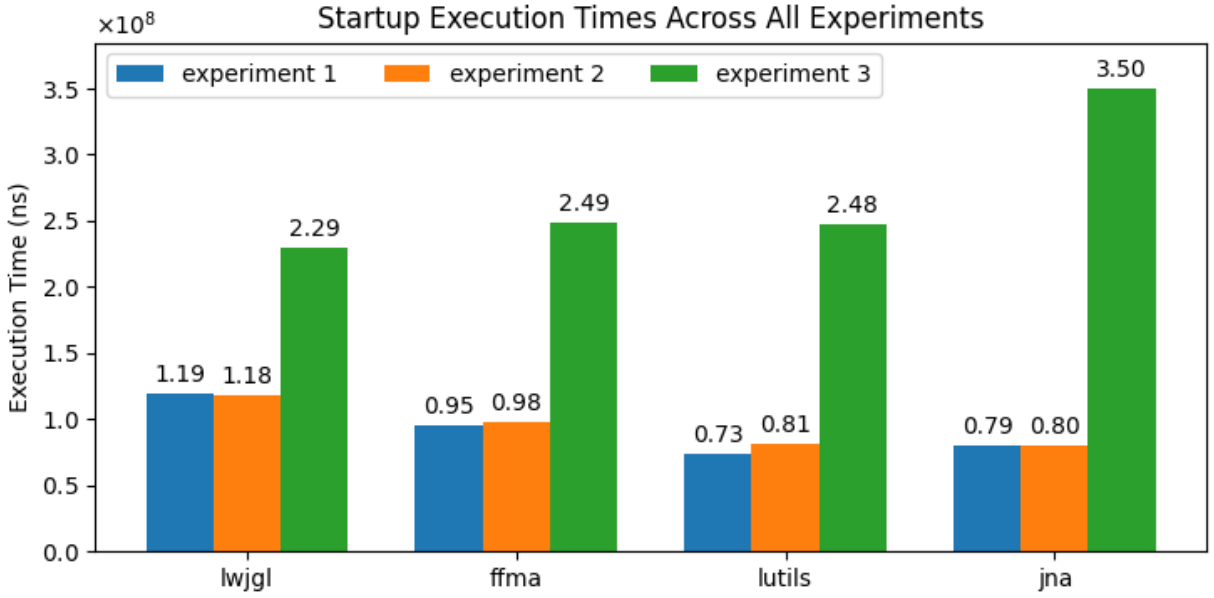


Fig. 12: The execution times of benchmark 3 of each experiment. The reference values have been subtracted to isolate startup overhead.

| Library | LWJGL | FFMA | LUtils | JNA |
|---|---|---|---|---|
| Exp. 3 (ns) | 229,391,244 | 248,829,148 | 247,704,301 | 349,907,362 |
| Exp. 2 (ns) | 117,634,620 | 97,781,989 | 81,092,109 | 80,251,941 |
| Exp. 1 (ns) | 118,695,278 | 94,844,143 | 73,165,734 | 79,408,842 |
| Abs. diff: Exp. 3 – Exp. 2 (ns) | 111,756,624 | 151,047,159 | 166,612,192 | 269,655,421 |
| Abs. diff: Exp. 3 – Exp. 1 (ns) | 110,695,966 | 153,985,005 | 174,538,567 | 270,498,520 |
| Ratio: Exp. 3 / Exp. 2 | 1.95 | 2.54 | 3.05 | 4.36 |
| Ratio: Exp. 3 / Exp. 1 | 1.93 | 2.62 | 3.39 | 4.41 |

Tab. 10: The execution times of benchmark 3 of each experiment. The reference values have been subtracted to isolate startup overhead. Additional calculations are displayed to highlight the startup overhead increase in experiment 3.

Overall, this benchmark shows once again, that the one-time initialisation and per-structure setup costs introduce more than 60% overhead across all libraries in terms of execution time. In Contrast, the allocation rate only increases by a factor smaller than two across all four libraries.

### 4.4.6 Benchmark 3 Interpretation

Assuming that across all three experiments each library requires the constant individual execution time for one-time initialisation (in all instances of benchmark 3), column 5 of Tab. 10 should show part of the overhead introduced by the one-time setup required for the large and complex structure. Specifically, the startup overhead introduced by the large and complex structure minus the startup overhead introduced by the two small structures of experiment 1. This means that startup overhead of the large and complex structure is on average at least $111 \cdot 10^6$ ns for LWJGL, followed by FFMA and LUtils at $154 \cdot 10^6$ ns and $175 \cdot 10^6$ ns respectively. JNA introduces a higher startup overhead with $270 \cdot 10^6$ ns. However, these values must be interpreted with caution, because it is unknown how much of the execution time is spent for class loading, calculating the structure layout and potentially other unknown factors. Furthermore, startup overhead for a second structure might be smaller, because some classes are already loaded. For example, classes for arrays of structures (`StructureArray`) or the wrapper classes for primitive types from LUtils (`BBInt1`, `BBFloat1`, …).

In contrast to the results of benchmark 3 from the previous experiments these values suggest that across all four libraries approximately half of the execution time is spent executing per-structure initialisations. This contradicts the claim from previous experiments, that per-structure initialisation cost is relatively small compared to the dominant one-time initialisation overhead.

In conclusion, to precisely analyse the startup overhead of the evaluated libraries more benchmarks would be required.

## 4.5 Summary

The startup cost benchmarks (benchmark 3) display that LUtils performs the best regarding execution time across experiment 1 and 2, followed by JNA and FFMA. LWJGL exhibits the worst performance in the first two experiments. However, in experiment 3 - evaluating large and complex structures – LWJGL displays the best startup performance, followed by LUtils and FFMA. JNA performs the worst. The allocation rate results show a different trend. LUtils demonstrates the best allocation performance in experiment 1 and 2, followed closely by JNA and LWJGL. FFMA exhibited the worst allocation rate in the first two experiments. Once again, the third experiment yields contrasting results: LWJGL achieves the lowest allocation rate,

followed by FFMA and LUtils, while JNA – consistent with the execution time measurements – exhibits the highest allocation rate in experiment 3.

The results of the other benchmarks show that LWJGL provides the best performance for allocating structures and write/read operations across different structure sizes. Following LWJGL, FFMA displayed the second-best performance, especially when allocating structures without performing write/read operations. However, FFMA performed worse when working with large and complex structures. JNA generally displayed the worst performance especially when operating with larger structures.

LUtils generally performed worse than LWJGL and FFMA but better than JNA. Only in the last experiment – evaluating large and complex structures with many arrays – LUtils performed better than FFMA in terms of execution time. Furthermore, the results show that LWJGL performed up to 83 times better than LUtils in terms of execution time, suggesting that LUtils should not be used for performance-critical applications, like real-time rendering. That is why the next chapter analyses bottlenecks and presents possible optimisations to increase performance.

# 5 Profiling and Optimisations

After completing the comparative evaluation of LWJGL, FFMA, JNA and LUtils, a profiling analysis is conducted to identify potential optimisations within LUtils. Since the evaluated benchmarks consist of relatively short-running code, accurate profiling requires a sampling profiler that does not suffer from the safepoint bias. Thus, the Java Flight Recorder and the Async-Profiler – introduced in section 2.3.3 – can be used. Since the benchmarks are executed on Linux, this work uses the Async-Profiler. Based on the profiling results problematic code is identified and possible optimisations discussed. Before analysing the results, the thesis highlights how the benchmarks are executed in combination with the Async-Profiler.

## 5.1 Benchmark Execution with Async-Profiler

As mentioned in 4.1 the fat-jar has two additional optional parameters, to enable the Async-Profiler and to profile allocation rate instead of execution time. If the async profiler is enabled, it is added to the JMH `OptionsBuilder` as seen in Code 13.

```
1  ChainedOptionsBuilder opt = new OptionsBuilder();
2  …
3  opt.addProfiler(AsyncProfiler.class, "<async-profiler-args>");
```

Code 13: Add the Async-Profiler using JMH's OptionsBuilder.

The placeholder `<async-profiler-args>` is replaced with several arguments:

- `libPath` is set to the path of the native library required by the Async-Profiler. This DLL file is automatically extracted to a temporary path by the fat-jar at runtime
- `output` is set to `flamegraph`, which tells the async profiler to generate an flamegraph as HTML visually showing which method calls required execution time or allocation rate.
- `simple` is set to `true`, which tells the async profiler to use simple class names without the package, making it easier to read.
- `dir` is set to `benchmark-results/` extended by the output directory passed as second argument to the fat jar. This specifies the output directory for flamegraph.
- `alloc` is set to `1k`, which tells the async profiler how often to sample the allocation space on average. This is intentionally set to a small value to get a precise measurement. The performance impact does not matter, because we measure execution time in a separate execution.

The fat jar itself is once again executed using a shell script as seen in Code 14.

```
1  OutputDir="$(date +"%Y-%m-%d_%H-%M-%S")"
2
3  java \
4    -Xmx8g
5    -XX:+UnlockDiagnosticVMOptions \
6    -XX:+DebugNonSafepoints \
7    -jar ./benchmark-runner.jar \
8    "lutils.<benchmark-name>" "$OutputDir" "true" "<enable-alloc>"
```

Code 14: Shell script to run a LUtils benchmark with the Async-Profiler. The placeholder "<benchmark-name>" must be replaced with the class name of the benchmark to execute. Furthermore, the placeholder "<enable-alloc>" must be replaced with a boolean, depending on whether execution time or allocation rate should be profiled.

After the shell script is executed, it creates a flamegraph as HTML which can be analysed to identify code that requires large amounts of execution time or allocation space.

## 5.2 Problem Identification

The flamegraph analysis provides detailed insight into the internal execution behaviour of LUtils. While the earlier benchmark results quantify performance differences between the evaluated libraries, profiling enables the identification of concrete code responsible for the observed execution times and memory allocation rates. This section summarises the performance bottlenecks revealed by the Async-Profiler and discusses potential optimisation strategies.

The Async-Profiler has been executed on benchmark 1 and 2 of each experiment. Benchmark 3 is run in single shot mode and thus cannot be reasonably analysed using the Async-Profiler, because the benchmark and its corresponding reference benchmark are executed together.

In the following subsections the profiling results of benchmark 1 and 2 of each experiment are summarised, and potential optimisations are mentioned shortly. During these subsections the generated flamegraphs are referenced. Some of these are included in Attachment A 1. Furthermore, all flamegraphs are included in the GitHub[27] repository as HTML. The next chapter collects possible optimisations and analyses them further.

### 5.2.1 Experiment 1

Benchmark 1 allocates and creates ten different small structures. Benchmark 2 allocates two different small structures and performs write and read operation on these. Despite the differences in workload the flamegraphs created by the benchmark 1 and 2, show a similar trend.

[27] https://github.com/lni-dev/thesis-lutils/tree/master/benchmark-results/profiling_results_no_forks/

These flamegraphs are also included in Attachment A 1 as Fig. 15: Flamegraphs for benchmark 1 (using LUtils) of experiment 1 generated by the Async-Profiler.Fig. 15 and Fig. 16.

*Native Memory Allocation*

In both benchmarks, native memory allocation via `BufferUtils.create64BitAligned()` is the one of the largest contributors to execution time at 36% and 26% respectively. This method internally relies on the Java class `ByteBuffer`. More specifically on the method `ByteBuffer.allocatedDirect()`. This could be optimised by using a faster native memory allocator. Possible candidates include FFMA's `Arena` and the `malloc` function from the C-standard-library or from other libraries like jemalloc[28] or rpmalloc[29].

*ByteBuffer Claiming and Slicing*

Another major contributor is the `Structure.claimBuffer()` method with approximately 24% in benchmark 1 and 26% in benchmark 2. This method calls `Structure.useBuffer()` of every structure element as well as on itself. Inside that method the Async-Profiler blames three different methods: `DirectByteBuffer.slice()`, `Structure.setInfo()` and `ComplexStructure.getChildren()`. The latter method requires about 4% execution time of benchmark 1, but it also appears outside of `Structure.claimBuffer()` resulting in a combined execution time of 15%. `ComplexStructure.getChildren()` retrieves the Java fields of each structure element using reflection. This is optimisable by calling the `init()` method and passing the variables of all structure elements in the correct order. However, this must be handwritten by the developer for every structure class.

A lot of the remaining ~20% of `Structure.claimBuffer()` are shared by `DirectByteBuffer.slice()` and `Structure.setInfo()`. The latter shares about as much blame as the former. However, this is not a realistic result, given that the code of `Structure.setInfo()` is relatively simple. Since both methods have been inlined, it is more likely that `Structure.setInfo()` is wrongfully blamed due to the inlining problem discussed in 2.3.2. This is further supported by the allocation profile, which shows that about one third of all allocated bytes are due to the `DirectByteBuffer.slice()` method. An alternative approach is to avoid slicing the `ByteBuffer` for each structure element. Instead, each element should store the same `ByteBuffer` instance and an offset.

[28] https://github.com/jemalloc/jemalloc
[29] https://github.com/mjansson/rpmalloc

*Runtime Structure Info Resolution*

The profiling results further reveal that `SSMUtils.getInfo()` contributes significantly to the execution time at 21% and 11% of benchmark 1 and 2 respectively. The high execution time of this method is due to the heavy use of reflection and retrieving of annotations at runtime. The two biggest contributors are the method calls `SSMUtils.getABI()` and `SSMUtils.sanityChecks()`.

The former retrieves the ABI from the `StructureLayoutSettings` annotation of the structure at runtime. The flamegraph reveals that every call to a method of an annotation translates to a lookup in a `LinkedHashMap`, which requires a lot of execution time. Furthermore, the lookup for the annotation itself is also performed using a hash map. A possible optimisation would be to require the developer to pass a reference to the ABI in the constructor of the structure. This would eliminate the `SSMUtils.getABI()` method and the `StructureLayoutSettings` annotation completely.

The latter method `SSMUtils.sanityChecks()` shows a similar pattern, consisting only of hash map lookups due to annotations. Specifically, this method checks whether the `StructureSettings` annotation of the structure to be created allows the configuration set by the developer. Consequently, this code is not required for any logic but purely to improve error detection. One possible optimisation would be to disable this error checking in production environments.

The rest of the `SSMUtils.getInfo()` consists of checking the `StructureSettings` annotation whether this structure has a structure `INFO` variable or a `GENERATOR` variable – once again a hash map lookup. The most obvious optimisation is to remove the ability to provide a static structure `INFO` variable and enforce that the structure must provide a static structure `GENERATOR` method.

Furthermore, `SSMUtils.getInfo()` retrieves the `StructureInfo` from the `INFO` or a `GENERATOR` variable. For structures extending `ComplexStructure`, this consists of fetching the previously calculated structure info from a hash map. However, removing this hash map lookup by always recalculating the structure layout is not a plausible optimisation, because calculating the layout requires reflection and annotation retrieving – once again hash map lookups.

*Re-entrant Lock*

The allocation profiling results demonstrate that about 4% to 6% of the allocation space is used by a `ReentrantLock`. This can be optimised, because this lock is only required if modification tracking is enabled for the structure – which is not the case in any benchmarks. Modification tracking can only be enabled during constructor initialisation and is stored in a final variable. Thus, the `ReentrantLock` should not be created if modification tracking is disabled.

*Write and Read Operations*

Benchmark 2 shows that approximately 33% of the execution time was spent writing, reading or in code related to benchmark logic. The Async-Profiler claims that about 4% was due to writing and reading operations, but this should be interpreted with care, because each individual write and read operation is relatively short and often inlined – which may skid the profiling results due to the problems mentioned in 2.3.2. However, what is visible is the complex code path required for some write and read operations. These execution paths can be examined in greater detail in the flamegraphs of benchmark 2 from experiment 2. For instance, Fig. 13 illustrates the complete call chain triggered by a single `Int1.get()` invocation.

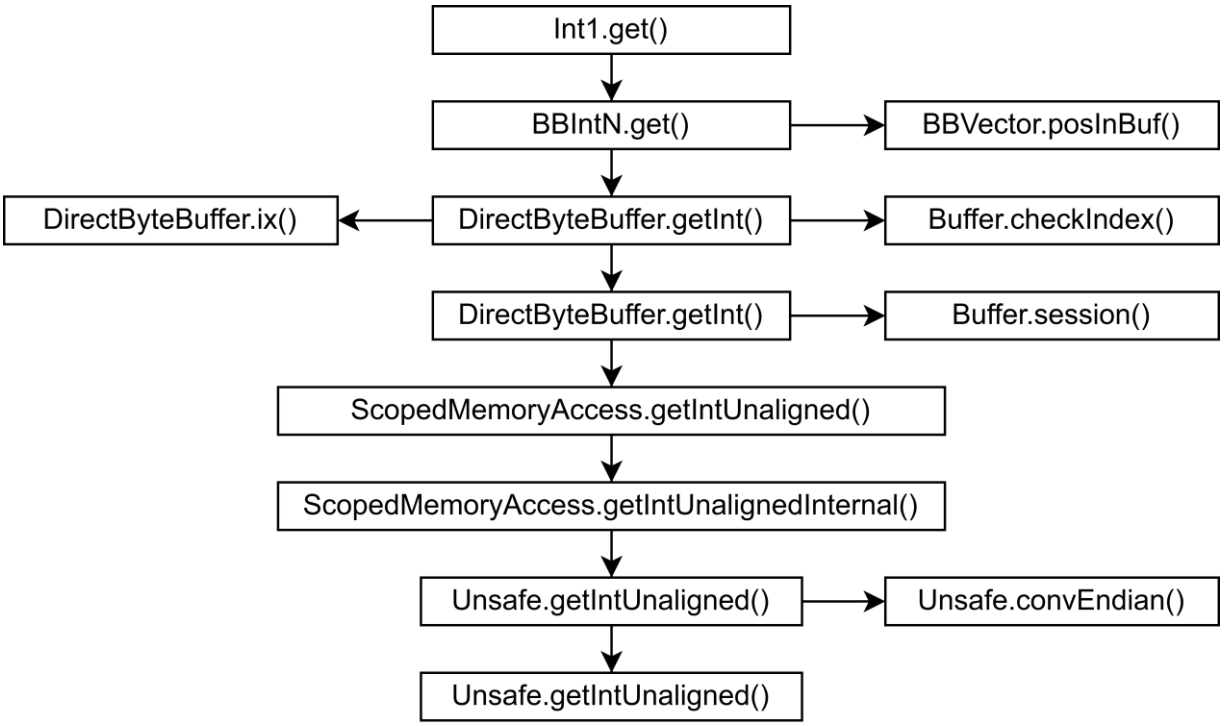


Fig. 13: Code path of an `Int1.get()` invocation.

One potential improvement is to perform write and read operations like LWJGL. Instead of using the respective `ByteBuffer.get*()` methods, calling `Unsafe.get*Unaligned` directly, avoids overhead introduced by Java's `DirectByteBuffer`. Analogously, write operations can be implemented using the corresponding `Unsafe.put*Unaligned`d methods.

### 5.2.2 Experiment 2

Benchmark 1 allocated and created five different complex structures. Benchmark 2 allocated a single complex structure and performed write and read operation on it. Once again, the flamegraphs created by benchmark 1 and 2 of experiment 2 show a similar trend despite the differences in workload. Additionally, they further validate the problems found during profiling of experiment 1.

*Native Memory Allocation*

The native memory allocation requires approximately 8% of execution time in benchmark 1 and 9% in benchmark 2.

*ByteBuffer Claiming and Slicing*

The execution time required to claim and repeatedly slice the `ByteBuffer` has increased in both benchmarks to 71% and 33% respectively. This bottleneck is further validated by the allocation profiling results, which show that 50% and 36% of bytes are allocated during these slicing operations.

*Runtime Structure Info Resolution*

In this experiment `SSMUtils.getInfo()` requires about 7% of execution time, which is less than in the previous experiment. Nevertheless, the methods invoked within `SSMUtils.getInfo()` remain identical to those in experiment 1 and exhibit a comparable distribution of execution time.

*Re-entrant Lock*

The `ReentrantLock` class is once again present in the allocation profile. However, it dropped to only 1% and 2% of allocated bytes in benchmark 1 and 2 respectively.

*Write and Read Operations*

Writing, reading and the benchmark logic itself required about 38% of execution time in benchmark 2. This time the Async-Profiler blames about half of it on writing and reading operations. Once again interpreting must be done with care, because each individual write and read operation is relatively short and often inlined – which may skid the profiling results due to the problems mentioned in 2.3.2. However, the complex code paths for write and read operations are clearly visible, as stated in the previous subsection.

### 5.2.3 Experiment 3

In this experiment benchmark 1 allocated and created five different large structures. Benchmark 2 allocated a single large structure and performed write/read operations. The flamegraph in Fig. 17 shows that benchmark 1 spends approximately 94% of its execution time in the java stub `jbyte_fill`[30]. Unfortunately, due to the nature of how stub code is executed, the async profiler cannot read a stack trace for this method. Only about 4% of execution time can be attributed to the benchmark. These 4% exhibit a similar distribution to benchmark 1 of experiment 2. This means that, the bottlenecks observed in the experiment 1 and 2 consume less than 4% of execution time in this experiment and are therefore not considered further. However, new bottlenecks appear.

*Structure Arrays*

The execution profile of benchmark 1 shows a bottleneck, which consumes about 96% of execution time but due to the missing stack trace it cannot identify the cause – apart form the name `jbyte_fill`. However, the source code of `jbyte_fill`[31] for ARM64 architectures suggests that it is used to fill arrays with a specific value. Looking at the allocation profiling results, it shows that about 80% of allocated bytes are required by `Structure` arrays. This suggests that `jbyte_fill` might be used to fill the memory required for these arrays with zeros. The large arrays are required by the `StructureArray` class. The constructor of `StructureArray` creates an array of `Structure` to potentially hold the `Structure` instance of each array element. Possible optimisations include the removal of the `Structure` array from the `StructureArray` class and instead creating a new instance whenever a child element is retrieved. Another possible optimisation is to maintain a single reusable instance of the child structure that is returned on each access, with its underlying byte buffer and offset reassigned as needed.

[30] https://github.com/openjdk/jdk/blob/d19eab4f08592140229de43689c7d20ff7fbf4ee/src/hotspot/share/runtime/stubRoutines.cpp
[31] https://github.com/openjdk/jdk/blob/497dca2549a9829530670576115bf4b8fab386b3/src/hotspot/cpu/aarch64/stubGenerator_aarch64.cpp#L2398

The second optimisation is further supported by the profile results of benchmark 2, which show that about 19% of execution time is spend inside the `StructureArray.get()` method. This method currently creates a new instance for the requested child and caches it in the `Structure` array for future retrieves.

*Write and Read Operations*

Writing, reading and benchmark logic requires approximately 81% of execution time in benchmark 2. This time the Async-Profiler blames about 24% of the benchmark's execution time on writing and reading operations. Once again this should be interpreted with care, because each individual write and read operation is relatively short and often inlined – which may skid the profiling results due to the problems mentioned in 2.3.2. However, an interesting observation is that `NativeInt32Array.get()` creates java's integer wrapper classes. Further inspection shows that this is a user error. The `NativeInt32Array` provides multiple get methods. Specifically, a normal get method which returns `Integer` and a `getInt` method which returns a primitive type integer. A better approach would be to let `get()` return a primitive type integer as well to avoid such user errors.

## 5.3 Optimisations

This section summarises and analyses the different optimisations introduced in the previous section.

### 5.3.1 Allocation strategy

During experiment 1 the native memory allocation required more than 26% of execution time. Therefore, using a faster allocator should improve performance drastically, especially when smaller structures are created. LUtils currently uses `ByteBuffer.allocateDirect()`. Two alternatives are compared in this subsection. Specifically, `Arena` from FFMA, the C method `malloc` and `ByteBuffer.allocatedDirect()` which LUtils currently uses. To compare these an additional benchmark was run for each allocator.

Each allocator is executed with 175 different allocation sizes. These sizes are generated deterministically using a fixed random seed as seen in Code 15. Specifically, allocation sizes are derived from a logarithmic distribution resulting in allocation sizes that span from $2^5$ to $2^{19}$ bytes while emphasising smaller sizes. Furthermore, odd values are rounded up to the next even number to provide more realistic allocation sizes.

```
1  Random random = new Random(2989);
2  IntStream params = random.doubles(180, 5.0, 19.0)
3          .mapToInt(v -> {
4              int i = (int) Math.pow(2, v);
5              return (i % 2) == 0 ? i : i+1;
6          }).distinct().sorted();
```

Code 15: Random allocation size generator code. It Generates 175 logarithmically distributed allocation sizes, which are used for the benchmark executed on the allocators of LUtils, FFMA and malloc.

The benchmark is configured to measure average execution time. For each allocation size a short warmup is included to allow the JIT compiler to stabilise, followed by three measurement iterations to obtain representative results. The configuration is summarised below:

- Benchmark mode: Average Execution time.
- Warmup: two warmup iterations. Three seconds per iteration.
- Measurement: three measurement iterations. Three seconds per iteration.

Additionally, garbage collection is explicitly enabled via JMH's option `shouldDoGC(true)` to ensure that the deallocation of native memory does not accumulate across iterations. This is only important for `ByteBuffer.allocateDirect()`, because the native memory of FFMA and malloc is freed explicitly inside the benchmark. This also means that realistic execution times for `ByteBuffer.allocateDirect()` are higher than what is suggested by the results of the benchmark. However, this effect can be ignored compared to the already much higher execution times emitted by this allocator.

Furthermore, the benchmark for the allocator malloc is run twice. Once to only allocate memory and once to allocate memory and additionally zero-initialise the memory using `Unsafe.setMemory()`. In the following paragraphs malloc always refers to the second benchmark which zero-initialises the memory unless explicitly stated otherwise.

The results are displayed in Fig. 14. The sub figure shown in the zoomed view on the left displays small allocation sizes between 32 bytes and 2,136 bytes. In contrast, the sub figure on the right shows the full range of allocation sizes from 32 bytes to 515,724 bytes on a logarithmic x-axis.

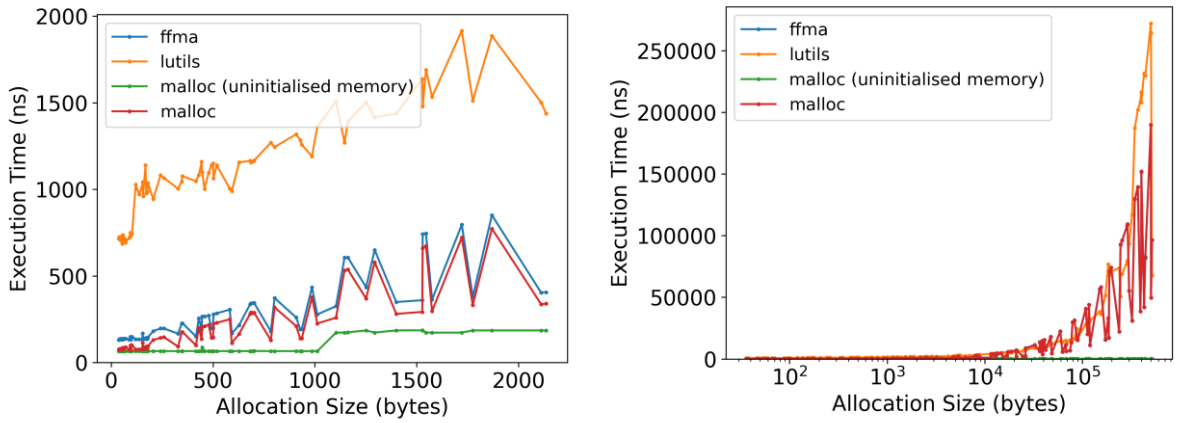


Fig. 14: Allocator comparison results. Measurements of the benchmark comparing the allocators of FFMA, LUtils and malloc. Left: Zoomed in to allocation sizes between 32 and 2,136 bytes. Right: Execution times of all allocation sizes between 32 and 515,724 bytes on a logarithmic x-axis.

Overall, the results demonstrate that malloc consistently exhibits a slightly better performance than FFMA. The execution time of malloc ranges from 73 ns to 189,797 ns with an average standard deviation of only 20.0 ns. FFMA's execution times range from 131 ns to 190,048 ns with an average standard deviation of approximately 19.9 ns. The low standard deviations show that the extreme variations in execution time from one allocation size to the next are introduced by the allocation size and not by random effects. Comparison between malloc without zero-initialised memory and malloc with zero-initialised memory show that these variations stem from `Unsafe.setMemory()`. LWJGL and FFMA clearly outperform LUtils up to allocation sizes of approximately $10^4$ bytes. LUtils - using `ByteBuffer.allocateDirect()` - demonstrates execution times ranging from 686 ns to 271,990 ns with the largest average standard deviation of 1,365 ns. It is important to mention is that all libraries demonstrate a linearly increasing execution time, even though the right sub figure wrongfully suggests an exponential increase due to the logarithmic x-axis.

The results indicate that native memory allocation of sizes less than $10^4$ bytes is faster via malloc than with Java-based alternatives. This suggests that replacing the current allocation strategy in LUtils with malloc could yield to substantial performance improvements, especially when allocating many small structures. For large ($> 10^4$ bytes) structures LUtils can fallback to `ByteBuffer.allocateDirect()`, which provides a more predictable allocation performance. Alternatively, using malloc provides the additional benefit that filling the memory with

zeros is optional. if memory clearing is not required malloc provides an almost constant allocation performance ranging from 65 ns to 185 ns and a standard deviation of 0.09 ns.

To summarise, allocations of sizes less than $10^4$ bytes should be done via malloc. Allocation sizes larger than $10^4$ bytes should be allocated using `ByteBuffer.allocateDirect()` unless memory clearing is not required. In this case malloc once again provides the best performance.

However, even better performance could be achieved if allocations would not happen at all. Especially during render loops which repeatedly allocate the same structures, the allocation time might be a problem. This could be solved by allocating a large chunk of memory in advance and using it like a stack. As a matter of fact, this has already been implemented in LUtils and can be used as seen in Code 16.

```
1  // Allocate memory once
2  Stack stack = new DirectMemoryStack64();
3
4  // Enter render loop
5  while(true) {
6      try(var _ = stack.popPoint()) { // create closeable pop point
7          SmallTestStruct2 aStruct = stack.push(new SmallTestStruct2());
8          BBInt1 aInt = stack.pushInt();
9          …
10     } // all structures are automatically popped once close() is called.
11 }
```

Code 16: Using a stack to eliminate per-frame allocations. Example usage of LUtils' `DirectMemoryStack64` to avoid repeated allocations.

### 5.3.2 Runtime Structure Child Element Resolution

A contributor with less than 15% in all experiment is the call to `ComplexStructure.getChildren()`, which retrieves all structure elements using reflection and stores them into the array called `items`. The optimisation is already available but must be manually "activated" by the developer, by passing all structure elements in the correct order to the `init()` call inside the constructor. This is user error prune and tidies. This could be solved by providing a structure class generator like LWJGL's Generator. This generator could accept the structure definition as input either in C-code or in a domain specific language (DSL). The output is the generated java class for a given structure, including the call of `init()` in the constructor with all structure elements passed in the correct order.

Additionally, Further investigation reveals that said array `items` is only required to call `useBuffer()` of all child elements when `useBuffer()` of the parent structure is called. This

means that the generator can hardcode the `useBuffer()` method and remove the `items` variable resulting in a reduction of required Java-Heap memory. The current `useBuffer()` code seen in Code 17 could then look like Code 18 for a structure with two elements called `aFloat` and `aInt`.

```
1  @Override
2  protected void useBuffer(
3          @NotNull Structure m, //mostParentalStructure
4          int offset,
5          @NotNull StructureInfo info
6  ) {
7      super.useBuffer(mps, offset, info);
8      if(items == null)
9          return;
10     ComplexStructureInfo cInfo = getInfo();
11     StructVarInfo[] childrenInfos = cInfo.getChildrenInfo();
12     int[] sizes = cInfo.getSizes();
13
14     int position = 0;
15     for(int i = 0; i < items.length; i++) {
16         position += sizes[i * 2];
17         if(items[i] != null)
18             items[i].useBuffer(m, offset + position childrenInfos[i].getInfo());
19         position += sizes[i * 2 + 1];
20     }
21 }
```

Code 17: LUtils current code of the `useBuffer()` method of a `ComplexStructure`. It calls `useBuffer` of every child element.

```
1  @Override
2  protected void useBuffer(
3          @NotNull Structure m, //mostParentalStructure
4          int offset,
5          @NotNull StructureInfo info
6  ) {
7      super.useBuffer(m, offset, info);
8      ComplexStructureInfo cInfo = getInfo();
9      StructVarInfo[] childrenInfos = cInfo.getChildrenInfo();
10     int[] sizes = cInfo.getSizes();
11
12     int position = sizes[0];
13     aFloat.useBuffer(m, offset + position, childrenInfos[0].getInfo());
14     position += sizes[1] + sizes[2];
15     aInt.useBuffer(m, offset + position, childrenInfos[1].getInfo());
16 }
```

Code 18: `useBuffer()` code optimisation through machine generated code. LUtils `useBuffer()` code could look like this if it was machine generated, removing the need for the `items` variable. In this example. the structure has two elements called `aFloat` and `aInt`.

### 5.3.3 Custom Buffer Implementation

The profiling results of experiment 1 and 2 show that more than 30% of all allocated bytes are due to slicing of Java's `ByteBuffer`. Instead of slicing the byte buffer, every child element could store the same byte buffer instance and an offset at which its memory starts.

The problem with this optimisation is that Java's `ByteBuffer` has an internal state. For example, its limit (size) can be changed to any value smaller than the actual size. That is why a new buffer interface must be created, which only provides basic method and ensures that the buffers address and size are always constant.

The optimisation to replace `DirectByteBuffer`'s `get*()` and `put*()` methods with `Unsafe.get*Unalinged` and `Unsafe.put*Unalinged` similarly benefit from such a buffer interface. However, this optimisation introduces certain challenges – The `Unsafe` class is an JDK internal class, which may be removed or changed without notice. Nevertheless, it outperforms a custom native method using JNI for several reasons. Each JNI call requires additional work, including native stack setup and teardown as well as native exception checking [25]. Furthermore, JNI methods cannot be inlined by the JIT-compiler. Finally, the method from `Unsafe` is an intrinsic candidate[32], meaning that the JVM can replace it with handwritten assembly or byte code, thereby increasing performance [15]. Introducing a custom buffer interface as seen in Code 19 has the benefit that it can provide different underlying implementations. Meaning, that it is possible to fallback to a custom native method if `Unsafe` is not available.

```
1  public interface NativeMemBuffer {
2      long address();
3      long size();
4
5      byte getByte(long offset);
6      void setByte(long offset, byte value);
7
8      int getInt(long offset);
9      void setInt(long offset, int value);
10
11     // ... more get and set methods for short/long/float/double
12 }
```

Code 19: Custom buffer interface, which represents a chunk of native memory. It does not provide any way to change the address or the size of the memory. Meaning that these variables are constant.

[32] https://github.com/openjdk/jdk/blob/4e6cf8f5611b6f1ae1d18b01e95216d9bf43ee5a/src/java.base/share/classes/jdk/internal/vm/annotation/IntrinsicCandidate.java

To summarise this optimisation, during native memory allocation a new instance of `NativeMemBuffer` is created and passed to the structure with the offset zero. This structure then supplies all child elements with the same instance and their respective offset. Thus, avoiding slicing the buffer for every child element. Furthermore, all get and set methods can be implemented redundantly using `Unsafe` and custom JNI methods. During native memory allocation it is checked if `Unsafe` is available and the correct buffer implementation returned.

### 5.3.4 No ABI Resolution using Reflection

All subclasses of `ComplexStructure` perform the ABI resolution using reflection, this requires about 1% to 6% of execution time in experiment 1. Problematic is the resolution of the `StructureLayoutSettings` annotation and its related methods. As discussed in 5.2.1 each annotation and annotation method resolution translates to a hash map lookup at runtime. A promising optimisation is to remove the `StructureLayoutSettings` annotation completely and instead pass the desired ABI in the constructor of `ComplexStructure`. This results in the loss of multiple features:

1. A developer can no longer specify the ABI using the annotation
2. A developer can no longer specify a static method in the annotation which would be used to resolve the ABI at runtime.
3. A developer can no longer specify how the ABI of the child elements should be overwritten

However, these features are replaceable:

1. The developer now passes the ABI in the constructor
2. The developer can pass the return value of that method in the constructor
3. This is feature will be removed. However, it is an advanced use case, that structures with different ABIs are used in a single `ComplexStructure`, but it can be achieved if the developer implements a custom `useBuffer()` method.

The optimised `ComplexStructure` class would have an additional constructor parameter `abi`, which is stored in the class. Additionally, `usebuffer()` would also require an additional parameter `abi`. When `usebuffer()` is called and the stored ABI is `null` the ABI passed to `usebuffer()` would be used and stored. If `null` was passed to `useBuffer()` as well, the platforms default ABI would be used if available. In the case that an ABI is already stored and another ABI is passed to `useBuffer()`, an error would be thrown. The `useBuffer()` method would subsequently call `useBuffer()` on all child elements while passing the stored ABI.

### 5.3.5 Disable Validations in Production Environments

The method `SSMUtils.sanityChecks()` required about 3% to 8% of execution time in experiment 1 and 2. As the method name suggests, it only performs checks whether the structures' settings are valid. This can be optimised by not invoking the method in production environments. There are two possible ways to achieve this. On the one hand, the method can be wrapped in an assert statement, thus only executing if Java assertations are enabled. On the other hand, a static boolean variable can be provided, which allows developer to enable and disable these checks as needed.

### 5.3.6 Further Reduce Annotation Lookups

The runtime structure info resolution required about 7% to 21% of execution time in experiment 1 and 2. Many contributors have already been tackled in the last two optimisations. However, the `StructureSettings` annotation still contains one annotation method which is checked outside of the `SSMUtils.sanityChecks` method. Specifically, the method `requiresCalculateInfoMethod()`, which allows a structure to specify whether the info for this structure is available in a `static final` variable or must be calculated through a `StaticGenerator` – whose instance must also be saved in a `static final` variable. This can be optimised by always requiring the `StaticGenerator`. If the `StructureInfo` of a structure is constant – which is the only use case of a storing the info directly into a `static final` variable – the `StaticGenerator` can return the constant `StructureInfo` instance.

### 5.3.7 More Structure Array Implementations

Approximately 80% of allocated bytes in benchmark 1 of experiment 3 are caused by Structure arrays required by the `StructureArray` class. That is because the `StructureArray` class creates these arrays to hold the `Structure` instances of each child element. This is fine if the elements are retrieved repeatedly and Java-Heap memory is not a problem. However, additional options should be provided to the developer:

- Allow the developer to pass a parameter to the `StructureArray` which disables the initialisation of the Structure arrays, thereby disabling the caching of structure instances for child elements. If this is disabled each `get()` call returns a new `Structure` instance.
- Allow the developer to pass a structure instance, obtained from a previous `get()` call, to subsequent `get()` invocations. The `get()` method would then call `useBuffer()` on the passed structure instance, thereby overwriting the native memory region the `Structure` instance is backed by.

# 6 Summary

This thesis introduces the newly developed library LUtils to work with structures while providing different Application Binary Interfaces (ABI). To determine whether LUtils represents a viable alternative to existing solutions in terms of execution time and memory behaviour, several Java native access libraries were evaluated and compared – namely LWJGL, FFMA, JNA and the introduced library LUtils.

To achieve this, multiple JMH (Java Microbenchmark Harness) benchmarks were designed to measure structure creation and write/read operations. These benchmarks measure execution time as well as Java-Heap allocation rate and thereby reveal performance characteristics. LWJGL demonstrated the best performance, followed by FFMA which displays low execution times for small structures, but decreasing performance with large and complex structures. LUtils and especially JNA display generally lower performance, likely due to their use of Java reflection.

The evaluation revealed that LUtils should not be used in performance critical applications like real-time rendering, because LWJGL provides up to 83 times better performance. That is why the benchmarks executed for LUtils have been profiled using the Async-Profiler. The profiling analysis uncovered optimisation opportunities regarding allocation strategies, reducing reflection usage, write/read operations and more. By implementing these optimisations LUtils might become a viable option for performance-critical Java applications.

Overall, the thesis provides a comparison of the Java native access libraries LWJGL, FFMA, JNA and LUtils, as well as an in-depth profiling analysis of LUtils, and thereby deriving possible optimisations.

# 7 Outlook

The results of this thesis provide a performance comparison of the evaluated libraries – LWJGL, FFMA, JNA and LUtils – with respect to structure creation, memory allocation and read/write operations. While the evaluation reveals clear performance characteristics, bottlenecks and possible optimisations for LUtils, several aspects remain open for further investigation. This chapter outlines directions for further research.

First, a re-evaluation of LUtils after implementing the optimisations proposed in section 5.3 would allow a quantitative assessment of their effectiveness. Such a follow-up study could determine the extent to which the identified bottlenecks influence overall performance and whether the proposed improvements lead to measurable gains.

In addition, several aspects that were not examined in this thesis warrant further analysis. All benchmarks were conducted on Pop!_OS using the same hardware configuration. Since performance characteristics may depend on the operating system or processor architecture, experiments on additional platforms – such as windows or alternative hardware – could result in different observations regarding performance.

Furthermore, benchmark 2 of each experiment performed write and read operations on all fields of the structures. However, in practice applications might modify only a subset of structure fields. Investigating partial write/read operations may therefore reveal different performance characteristics.

Another important aspect concerns one-time initialisation costs. When a structure is used for the first time, all evaluated libraries introduce performance overhead due to one-time initialisations such as layout calculation and class loading. Although such costs are implicitly included in the results of Benchmark 3, they cannot be clearly distinguished from one-time per-application initialisation overhead. Moreover, because Benchmark 3 could not be analysed using the Async-Profiler, execution time could not be attributed to specific methods. Therefore, a more detailed investigation of startup behaviour represents a promising direction for future research. In particular, performance increasing optimisations could be derived by performing method-level profiling on the initialisation phases of LUtils as well as LWJGL, FFMA, and JNA, all of which exhibit significant startup overhead.

Another opportunity for further research is the performance impact of using different allocation strategies. While this thesis analysed the time required to allocate the native memory, future work could evaluate the influence of different allocators on the performance of complete

applications. For instance, although a particular allocator may require more execution time during allocation, it could improve subsequent write/read performance due to cache locality. Thus, resulting in a better overall execution time if enough write or read operations are performed. An evaluation of different allocators – including `Unsafe.allocateMemory`, malloc, jemalloc and rpmalloc – would provide deeper insight into this trade-off between allocation cost and long-term performance.

# List of Sources

[1] S. Liang, *The Java Native interface: programmer's guide and specification*. in The Java series. Reading, Mass: Addison-Wesley, 1999.

[2] A. Leonard, *Java Coding Problems: Become an expert Java programmer by solving over 250 brand-new, modern, real-world problems*, 1. Aufl. Birmingham: Packt Publishing Limited, 2024.

[3] G. Sellers, *Vulkan™ programming guide: the official guide to learning Vulkan*. in Always learning. Boston Munich: Addison-Wesley, 2017.

[4] A. Munshi, „The opencl specification", in *2009 IEEE Hot Chips 21 Symposium (HCS)*, IEEE, 2009, S. 1–314. Zugegriffen: 11. Februar 2026. [Online]. Verfügbar unter: https://ieeexplore.ieee.org/abstract/document/7478342/

[5] Y. Li, T. Tan, und J. Xue, „Understanding and Analyzing Java Reflection", *ACM Trans. Softw. Eng. Methodol.*, Bd. 28, Nr. 2, S. 1–50, Apr. 2019, doi: 10.1145/3295739.

[6] I. R. Forman, *Java reflection in action*. 2005. Zugegriffen: 11. Februar 2026. [Online]. Verfügbar unter: https://www.lirmm.fr/~dony/enseig/MRStock/notes-etudes/Java_Reflection_in_Action_2005.pdf

[7] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, und N. Nystrom, „Use at your own risk: the Java unsafe API in the wild", *SIGPLAN Not.*, Bd. 50, Nr. 10, S. 695–710, Dez. 2015, doi: 10.1145/2858965.2814313.

[8] H. J. Lu, M. Matz, J. Hubicka, A. Jaeger, und M. Mitchell, „System V application binary interface", *AMD64 architecture processor supplement*, S. 588–601, 2018.

[9] M. Rodriguez-Cancio, B. Combemale, und B. Baudry, „Automatic microbenchmark generation to prevent dead code elimination and constant folding", in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, S. 132–143. Zugegriffen: 13. Oktober 2025. [Online]. Verfügbar unter: https://ieeexplore.ieee.org/document/7582752/

[10] H. H. Karer und P. B. Soni, „Dead code elimination technique in eclipse compiler for Java", in *2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, Dez. 2015, S. 275–278. doi: 10.1109/ICCICCT.2015.7475289.

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, und P. F. Sweeney, „Evaluating the accuracy of Java profilers“, in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto Ontario Canada: ACM, Juni 2010, S. 187–197. doi: 10.1145/1806596.1806618.

[12] H. Burchell und S. Marr, „Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy through Machine Code-Level Slowdown“, *Proc. ACM Program. Lang.*, Bd. 9, Nr. OOPSLA2, S. 3615–3641, Okt. 2025, doi: 10.1145/3763180.

[13] N. Wakart, „Profilers are lying hobbitses“, gehalten auf der JokerConf, St Petersburg, 2017. Zugegriffen: 11. Februar 2026. [Online]. Verfügbar unter: https://assets.ctfassets.net/oxjq45e8ilak/3vEi67mhZCqsygqqUqWWAq/ad670990989cdd3cf9763e7d6804ec90/Profilers_Are_Lying__Bastards-Joker.pdf

[14] D. A. Patterson und J. L. Hennessy, *Computer organization and design: the hardware/software interface*, ARM® edition. Amsterdam Boston Heidelberg London New York Oxford Paris San Diego San Francisco Singapore Sydney Tokyo: Elsevier, Morgan Kaufmann is an imprint of Elsevier, 2017.

[15] B. J. Evans, J. Gough, und C. Newland, *Optimizing Java: practical techniques for improving JVM application performance*, First edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly, 2018.

[16] A. Nisbet, N. M. Nobre, G. Riley, und M. Luján, „Profiling and Tracing Support for Java Applications“, in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, Mumbai India: ACM, Apr. 2019, S. 119–126. doi: 10.1145/3297663.3309677.

[17] F. Li, P. Agrawal, G. Eberhardt, E. Manavoglu, S. Ugurel, und M. Kandemir, „Improving memory performance of embedded Java applications by dynamic layout modifications“, in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, Apr. 2004, S. 159-. doi: 10.1109/IPDPS.2004.1303150.

[18] V. Sochat und T. Haines, „Binary-level Software Compatibility Tool Agreement“, 2022, *arXiv*. doi: 10.48550/ARXIV.2212.03364.

[19] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, und D. Wang, „Safe Java Native Interface“, in *ISSSE*, 2006.

[20] M. Grichi, M. Abidi, Y.-G. Guéhéneuc, und F. Khomh, „State of practices of Java native interface“, in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, in CASCON '19. USA: IBM Corp., Nov. 2019, S. 274–283.

[21] G. Tan und J. Croft, „An Empirical Security Study of the Native Code in the JDK.“, in *Usenix Security Symposium*, 2008, S. 365–378. Zugegriffen: 13. Februar 2025. [Online]. Verfügbar unter: https://www.usenix.org/event/sec08/tech/full_papers/tan_g/tan_g_html/

[22] N. A. Tomilov und V. P. Turov, „Evaluating the performance of Java Vector API in vector embedding operations“, *Computing, Telecommunications and Control*, Bd. 17, Nr. 4, S. 7–15, 2024.

[23] N. Seppälä, „Improving Java Performance With Native Libraries“, Metropolia University of Applied Sciences, 2024.

[24] L. Traini, V. Cortellessa, D. Di Pompeo, und M. Tucci, „Towards effective assessment of steady state performance in Java software: are we there yet?“, *Empir Software Eng*, Bd. 28, Nr. 1, S. 13, Jan. 2023, doi: 10.1007/s10664-022-10247-x.

[25] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, und K. Stoodley, „Inlining java native calls at runtime“, in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, Chicago IL USA: ACM, Juni 2005, S. 121–131. doi: 10.1145/1064979.1064997.

# Attachment A 1. Async-Profiler Flamegraphs

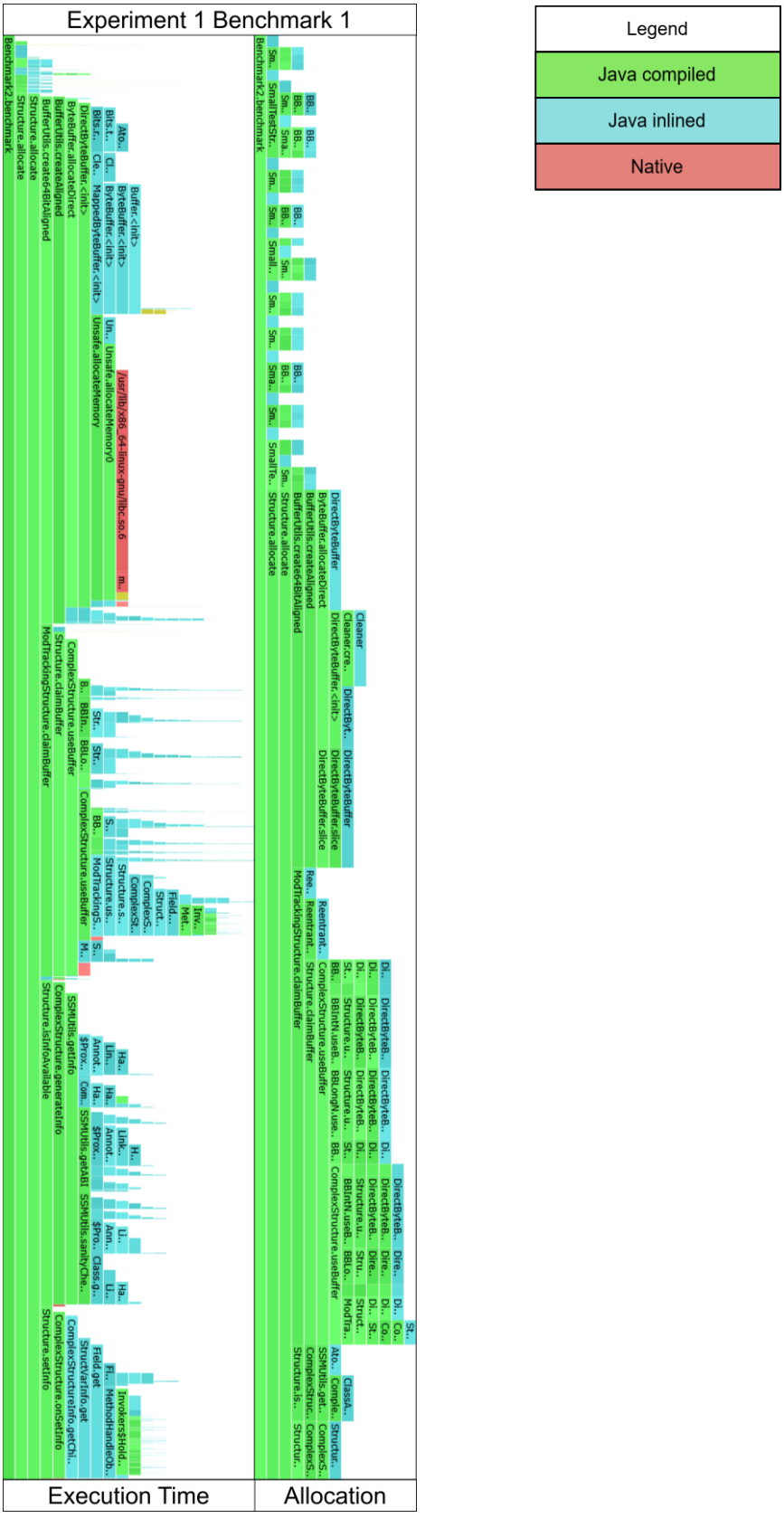


Fig. 15: Flamegraphs for benchmark 1 (using LUtils) of experiment 1 generated by the Async-Profiler.


Fig. 16: Flamegraphs for benchmark 2 (using LUtils) of experiment 1 generated by the Async-Profiler.


Fig. 17: Flamegraphs for benchmark 1 (using LUtils) of experiment 3 generated by the Async-Profiler.

## Use of AI Tools

In the preparation of this thesis, AI-based language models were used to improve wording, grammar, and overall readability of selected passages. The AI tools supported linguistic refinement and structural clarity but did not contribute to the scientific content, research design, data analysis, or interpretation of results. All technical concepts, experimental designs, evaluations, and conclusions presented in this thesis are the independent work of the author.