Comparison of Java vs. C++ Memory Management Performance

PETR LOPATKIN and BUGAYENKO YEGOR

The goal of this study is to investigate memory management performance in C++ and Java. We specifically investigate two research questions: (RQ1) when is Java's memory management playing better, and (RQ2) which factors explain the differences in performance of the two approaches. To do this, we explore a variety of synthetic benchmarks covering concrete use-cases for memory allocation, from fixed-size allocation, complex object construction, multi-threading, fragmentation, recursive allocation, to randomized memory access. Our findings indicate that while Java often benefits from an optimized garbage collection mechanism, certain allocation patterns in C++ can outperform Java in specific scenarios. The comprehensive analysis presented here provides insights for selecting appropriate memory management strategies in application development.

Contents

Authors' address: Petr Lopatkin; Bugayenko Yegor.

1 INTRODUCTION

There is one significant aspect in the design of high-performance applications that is memory management. In C++, developers utilize `malloc/free` functions (or the `new/delete` operator) to manually allocate and deallocate memory, while Java employs an automatic garbage collector to recycle memory. Though both languages possess various styles of memory management, the need for efficient heap allocation, deallocation, and fast memory access is a requirement they have in common.

Increasing complexity of modern applications and ever-growing performance demands have provoked active exploration of memory management techniques. Thus, the understanding of performance characteristics of manual and automatic memory management systems is essential for the selection of the appropriate technology for a specific application context.

2 RESEARCH QUESTIONS

In light of the above considerations, this study poses the following research questions:

RQ1 When is memory management in Java faster?

RQ2 Why is one memory management approach faster than the other?

3 RESEARCH METHOD

To answer these research questions, it was first necessary to examine current solutions for testing memory managers in these languages. After an extensive analysis, we identified the main and most popular benchmarks presented in Table 1. The benchmarks cover a variety of scenarios including fixed-size allocation, object allocation, multithreading, fragmentation, and recursive allocation.

Table 1. Benchmarking Suite Overview

Language	Source	Year of Activity	Description	Size (LoC)	Stars	Used By (Citations)	Version
C++	daanx/mimalloc	2019 –	Benchmarking suite for malloc implementations with various open-source benchmarks.	27025	382		1de7914
Java	dacapobench/dacapobench	2006–	Open-source Java non-synthetic benchmarks with real-world applications and memory-intensive loads.	464736	162	¸alves, 2021 Islam, 2023	59ab7e9
Java	renaissance-benchmarks /renaissance/	2019 –	A non-synthetic benchmark that uses third-party applications to load the language. Allows testing a wide range of jvm features	636121	312	Islam, 2023	6809835
C++	jeffhammond/STREAM	1995–2022	STREAM is the benchmark for measuring sustained memory bandwidth.	766	348		4dbce1d
C++	Adobe	2008 — 2008	Archive with various benchmarks, including some for memory usage, but limited suitability.	2671	N/A		December 8 2008 - second public release
Java	IonutBalosin/jvm-performance-benchmarks	2022–2024	Benchmarks for JVM including false sharing, scalar replacement, and more.	434630	90		857018d
Java	GarbageCollectorBenchmark	2015 — 2015	Synthetic benchmark for JVM garbage collector	127	0		a679ada

A number of relevant research projects underpin synthetic memory management testing for Java and C++. Importantly, (Daniel Haggander et al., 2001) provides C++ synthetic test case examples, while (Balosin, 2019) aggregates Java memory management benchmarks. These papers are significant reference points when understanding contemporary methodologies and making sense of existing benchmarking strategies.

3.1 Key Observations

Through an examination of existing benchmark suites, a range of key observations are made:

Synthetic Tests Availability: Java lacks numerous open-source synthetic memory management benchmarks, whereas C++ has significantly more such benchmarks.

Benchmarking Strategy: Java benchmarks tend to be more likely towards real-world uses or very specific use cases, whereas C++ benchmarks tend towards comprehensive synthetic tests that test distinct areas of memory management.

Framework Application: Benchmarking in Java relies quite extensively on the JMH framework, whereas C++ benchmarks function quite extensively without a proper framework.

Testing Mechanisms: Despite variation in implementation, the underlying testing mechanisms—i.e., heap allocation and read/write operations—are largely identical in both languages.

Relevance to Our Study: While the benchmarks being examined are handy for comparing performance of memory management, they were designed primarily for comparing specific implementations and not specifically for direct comparisons between languages. This is then a problem for comparing overall difference in performance when comparing C++ and Java.

To provide an equal and substantial comparison, we normalize all the test cases and organize them in a way to compare memory management performance under equivalent conditions. Through this, we can compare directly the efficiency of each approach unbiased by language-specific optimization.

All benchmarks utilized and related materials to this analysis along with references to previous work are available in the following repository:

Comparison-of-Java-vs.-C-Memory-Management-Performance.

4 PRELIMINARY EXPERIMENTS

Having set up the normalized benchmarking setup as above, the next job is to investigate the features of memory management in the two languages. This section brings to the forefront the prominent features of memory operations in Java and C++, setting the stage for our experimental study.

First of all, we need to understand the key aspects of memory management systems in every language. They provide support for three primitive types of memory - stack, heap and static area - that we will examine individually, also keeping in mind the details of their functioning in a multithreaded environment.

4.1 Features of memory operation in both languages

- **Stack Memory C++:**
  - A contiguous memory region allocated to each thread for storing local variables, function parameters, and return addresses (cppreference.com).
  - Each function call adds a new *stack frame* containing all its local variables (cppreference.com).
  - Frames are automatically removed upon function return (LIFO mechanism) (cppreference.com).
  - Stack size is limited, and overflow can occur due to deep recursion or excessive local variable usage (cppreference.com).
  - Each thread has its own stack, ensuring thread safety for local data (cppreference.com).

  Java:
  - Similar to C++, but managed by the JVM, with configurable stack size (Java VM Spec).
  - Local primitive variables and object references are stored in the stack, but actual objects reside in the heap (Oracle Docs).
  - Each thread has a separate stack, ensuring thread safety (Java Thread API).
- **Heap Memory**

  C++:
  - Used for dynamic memory allocation (`new`, `malloc`) (cppreference.com).
  - Allocated objects persist until explicitly deallocated (`delete`, `free`) (cppreference.com).
  - Automatic memory management does not cause the risk of memory leakage and fragmentation (isocpp.org).
  - Memory allocation can impact performance due to locking mechanisms in multithreaded programs.
  - Data races may occur when multiple threads access shared memory without synchronization.

  Java:
  - Objects are allocated in the heap and automatically managed by the garbage collector (GC) (Java VM Spec).
  - The heap is divided into the *Young Generation* and *Old Generation* to optimize GC performance (Oracle GC Tuning Guide).
  - Java ensures that freed objects are inaccessible, preventing use-after-free errors.
  - GC can introduce pauses, affecting real-time application performance (shipilev.net).
  - Synchronization primitives such as `synchronized`, `volatile`, and `ReentrantLock` help manage concurrent heap access.
- **Static Memory**

  C++:
  - Includes global and static variables, stored in the `.data` and `.bss` segments (cppreference.com).
  - Exists throughout the program's lifetime.
  - Requires explicit synchronization in multithreaded environments.

  Java:
  - The static variables reside in *Metaspace* (formerly *PermGen*) (Oracle Docs).
  - Persist until the class is unloaded by the JVM.
  - Improper use of static references can cause memory leaks (DZone Memory Leaks Guide).
  - Requires synchronization mechanisms such as `synchronized` or `volatile` for thread safety.
- **Memory Contiguity and Fragmentation**

  Memory Contiguity:
  - In C++, developers control memory layout: compact structures improve cache efficiency, while linked structures reduce locality (Agner Fog Optimization Guide).
  - Java places objects in the heap, with their location determined by the GC. Arrays of primitives are stored contiguously, but arrays of objects contain scattered references (Oracle GC Tuning Guide).

  Memory Fragmentation:
  - In C++, fragmentation occurs due to frequent allocations and deallocations of different-sized blocks. Custom allocators and memory pools mitigate this issue.

  - In Java, fragmentation is minimized by GC, which compacts memory, though fragmentation in the Old Generation may lead to long GC pauses (shipilev.net).
- **Multithreading Considerations**
  - Each thread has an isolated stack, ensuring local variable safety (Java Thread API).
  - Heap access requires synchronization to avoid data races:
    * C++: In C++, `malloc/free` can block threads if executed frequently.
    * Java: In Java, GC blocks memory allocation only in extreme cases, but accessing shared objects requires synchronization (shipilev.net).
  - Cache coherence issues arise in multicore systems, requiring proper memory visibility mechanisms Goetz, 2024.

4.2 Benchmarks

Based on all the theory above, we have formed the main benchmarks on which we will conduct our experiments:

- **Fixed Size Allocation**
  - *Purpose*: Measure performance of fixed-size memory allocation.
  - *Implementation*: Each iteration performs memory allocation of a fixed size
  - *Metrics*: Allocation time, deallocation time.
- **Complex Object Allocation**
  - *Objective*: To estimate memory allocation costs for complex and short-lived objects.
  - *Application*: Allocates memory for objects containing a vector (C++) or ArrayList (Java) field of the specified size.
  - *Metric*: Selection and deallocation time of primitive objects, Selection and deallocation time of complex objects.
- **Variable Size Allocation**
  - *Purpose*: Test memory allocation performance with varying block sizes.
  - *Implementation*: Allocation sizes are pre-generated by a Python script.
  - *Metrics*: Total time.
- **Concurrent Multithreaded Allocation Performance**
  - *Purpose*: Simulate real multitasking conditions for memory allocation
  - *Implementation*: Based on **Allocator bench**, but instead of a single-threaded execution, multiple threads perform memory allocations concurrently with different sizes
  - *Metrics*: Total time.
- **Fragmentation Test Random Allocation Release**
  - *Purpose*: Simulate and analyze memory fragmentation effects.
  - *Implementation*: Includes memory allocation and deallocation with varying sizes and random free patterns pre-generated by a Python script.
  - *Metrics*: Total time.
- **Recursive Memory Allocation**
  - *Purpose*: Test recursive memory allocation performance.
  - *Implementation*: Each iteration performs recursive memory allocation with a specified depth and size per level.
  - *Metrics*: Total time.

Experiments were conducted on native implementations and the Java JMH framework. All experiments were conducted with full optimization in C++ (i.e., with the `-O3` flag) and other configuration parameters specified in the respective project configuration files (i.e., `CMakeLists.txt` for C++ and `build.gradle.kts` for Java). For Java benchmarks, we used the JMH framework specifically to account for the unique characteristics of the Java Virtual Machine, including just-in-time (JIT) compilation and warm-up phases, ensuring more accurate and stable performance measurements. To ensure that our benchmarks accurately measure memory allocation and deallocation overheads, we made sure to prevent aggressive compiler optimizations from eliminating or altering our test cases. Modern compilers, such as GCC and Clang, will identify unused or redundant memory operations and optimize them away. To prevent this, we used several techniques, such as:

- Using `volatile` qualifiers and memory fences where appropriate to force the compiler to retain memory operations.
- Preventing escape analysis optimizations by ensuring allocated memory was used in ways that the compiler could not easily predict.
- Introducing artificial dependencies on allocated memory (e.g., performing dummy computations or writing results to externally visible structures).
- Using inline assembly or compiler-specific pragmas where necessary to prevent unwanted optimizations.

All source code, benchmark scripts, and detailed results are available in the repository:
Comparison-of-Java-vs.-C-Memory-Management-Performance.

5 CODE

Now that we have clearly defined our measurement criteria, we have developed the benchmark code. To offer deeper insight into its operation, we present the main portion of the source code, allowing you to verify its conditional consistency and understand its core functionality.

- **Fixed Size Allocation**

  The benchmark is run twice, just to check the allocation and de-allocation times the time measurement is put in the right place.

  C++:

```
for (size_t i = 0; i < iterations; ++i) {
    allocations[i] = new int8_t[allocation_size];
}

```

  Java:

```
for (int i = 0; i < iterations; i++) {
    byte[] allocation = new byte[allocationSize];
    allocations.add(allocation);
}
```

- **Complex Object Allocation**

  The **ComplexObject** class is an object that deals with an dynamically allocated int array. During the creation of the object, memory for the array is allocated, whose elements are filled with values equivalent to their indexes. Once the work with the object is finished, the memory for the array is released automatically by virtue of the built-in resource management mechanism.

  C++:

```
for (size_t i = 0; i < count; ++i) {
    buffer[bufferIndex] = new ComplexObject(complex_size);
    bufferIndex++;
    if (bufferIndex >= bufferSize) {
        for (size_t j = 0; j < bufferSize; j++) {
            delete buffer[j];
            buffer[j] = nullptr;
        }
        bufferIndex = 0;
    }
}
for (size_t j = 0; j < bufferIndex; j++) {
    delete buffer[j];
}
```

  Java:

```
for (int i = 0; i < iterations; ++i) {
    buffer[bufferIndex] = new ComplexObject(complexSize);
    bufferIndex++;
    if (bufferIndex >= bufferSize) {
        for (int j = 0; j < bufferSize; j++) {
            buffer[j] = null;
        }
        System.gc();
        bufferIndex = 0;
    }
}
```

- **Variable Size Allocation**

  C++:

```
for (size_t allocSize : allocationSizes) {
    int8_t* ptr = new int8_t[allocSize];
    allocations.push_back(ptr);
    for (size_t j = 0; j < allocSize; ++j) {
        ptr[j] = static_cast<int8_t>(j);
    }

}
for (int8_t* ptr : allocations) {
    delete[] ptr;
}
```

  Java:

```
for (int allocSize : allocationSizes) {
    byte[] arr = new byte[allocSize];
```

```
        allocations.add(arr);
        for (int j = 0; j < allocSize; j++) {
            arr[j] = (byte) j;
        }
    }
    allocations.clear();
    System.gc();
```

- **Concurrent Multithreaded Allocation Performance**

  C++:

```
    for (size_t i = 0; i < threadsNum; ++i) {
        threads.emplace_back([this, i, &allocationSizes]() {
            std::vector<int8_t*> allocations;
            allocations.reserve(allocationSizes.size());
            for (size_t allocSize : allocationSizes) {
                int8_t* ptr = new int8_t[allocSize];
                allocations.push_back(ptr);
                for (size_t j = 0; j < allocSize; ++j) {
                    ptr[j] = static_cast<int8_t>(i + j);
                }
            }
            for (int8_t* ptr : allocations) {
                delete[] ptr;
            }
        });
    }
    for (auto& t : threads) {
        t.join();
    }
```

  Java:

```
    for (int i = 0; i < threadsNum; ++i) {
        final int threadIndex = i;
        Thread thread = new Thread(() -> {
            ArrayList<byte[]> allocations = new ArrayList<>();
            allocations.ensureCapacity(allocationSizes.size());
            for (int allocSize : allocationSizes) {
                byte[] ptr = new byte[allocSize];
                allocations.add(ptr);
                for (int j = 0; j < allocSize; ++j) {
                    ptr[j] = (byte) (threadIndex + j);
                }
            }
            allocations.clear();
            System.gc();
        });
        threads.add(thread);
        thread.start();
    }
    for (Thread t : threads) {
        t.join();
    }
```

- **Fragmentation Test Random Allocation Release**

  C++:

```
    for (size_t i = 0; i < allocationSizes.size(); ++i) {
        allocations[i] = new int8_t[allocationSizes[i]];
        if (freePatterns[i]) {
            delete[] allocations[i];
            allocations[i] = nullptr;
        }
    }
```

  Java:

```
    for (int i = 0; i < allocationSizes.size(); ++i) {
        allocations.set(i, new byte[allocationSizes.get(i)]);
        if (freePatterns.get(i)) {
            allocations.set(i, null);
            System.gc();
        }
    }
```

- **Recursive Memory Allocation**

Here the **recursive_allocation** function I performs recursive allocation of memory of fixed size until it reaches the required depth.

C++:

```cpp
void recursive_allocation(size_t depth, size_t size, std::vector<int8_t*>& blocks, size_t index = 0) {
    if (index >= depth) return;
    int8_t* ptr = new int8_t[size];
    std::memset(ptr, 0, size);
    blocks[index] = ptr;
    recursive_allocation(depth, size, blocks, index + 1);
    std::memset(ptr, 1, size);
    std::memcpy(blocks[depth - 1], ptr, size);
}
recursive_allocation(depth, size, blocks);
for (int8_t* ptr : blocks) {
    delete[] ptr;
}
```

Java:

```java
private void recursiveAllocation(byte[][] blocks, int size, int index) {
    if (index >= blocks.length) return;
    byte[] ptr = new byte[size];
    Arrays.fill(ptr, (byte) 0);
    blocks[index] = ptr;
    recursiveAllocation(blocks, size, index + 1);
    Arrays.fill(ptr, (byte) 1);
    System.arraycopy(ptr, 0, blocks[blocks.length - 1], 0, size);
}
recursiveAllocation(blocks, allocationSize, 0);
for (int i = 0; i < depth; i++) {
    blocks[i] = null;
}
System.gc();
```

6 LIMITATIONS

While synthetic benchmarks provide a controlled environment in which to measure memory performance, you should keep a few limitations in mind before testing:

- **Synthetic Environment:** The benchmarks are based on controlled and often idealized conditions and perhaps do not encompass the full richness of actual applications. Field applications can have mixed workloads and uncontrolled behavior, e.g., unpredictable patterns of memory usage, and dynamic changing demands for resources.
- **Compiler and JVM Optimizations:** Performance measures are subject to significant variation according to C++ compiler optimizations and several varying JVM configurations in Java. For example, varying optimization levels, garbage collection modes, and run-time parameters might contribute to varying performance. Such factors tend to get optimized for single applications and would not usually carry over to everyone.
- **Test Coverage:** Although the benchmark suite exercises a wide range of memory allocation scenarios, it may not exercise all those encountered in production systems. Production systems may have complex interactions among deallocation, memory allocation, and garbage collection that are not fully replicated in synthetic testing.
- **Memory Leak Considerations:** In practice, issues such as memory leaks or poor resource handling may occur, which typically do not get simulated in artificial benchmarks. These issues may affect performance as well as stability but are based on ideal memory handling with no leaks in the benchmarks.
- **Memory Footprint Variability:** Memory size allocations used in the benchmarks are fixed or pre-specified by scripts. In real applications, memory footprints may be vastly smaller or greater, and the dynamic aspects of memory usage can influence allocation speed and efficiency in a different manner than in controlled benchmarking environments.
- **Hardware and Environmental Factors:** The benchmarks are generally performed with tested hardware and operating system configurations. Real-world deployments, on the other hand, may have heterogeneous environments with differing levels of hardware capability, different levels of system loads, and other outside factors affecting performance.
- **JVM Heap and Stack Size Settings:** For Java, the benchmarks included specifying the default heap and stack size parameters using JVM options to avoid memory overflows and stack overflow exceptions. By specifying larger maximum heap size and stack size, we gave sufficient resources to the JVM for handling large memory allocations without performance loss due to resource starvation. These configurations may be different from the typical memory settings used on smaller or less resource-intensive applications.
- **We can't call the garbage collector too often:** The garbage collector is designed to block all flows for safe garbage disposal. Because of this, if we call the garbage collector explicitly `System.gc()` too often, we will waste a very large amount of resources and time. For this reason, we had to introduce restrictions on how often the garbage collector is called.

7 RESULTS

Having run all benchmark tests with the provided configuration file, the following results were obtained. All benchmarks were run once, unless specified otherwise. The Java JMH benchmarks consisted of 3 warm-up iterations and 5 measurement iterations, these values were determined experimentally, and changing them did not yield a perceptible difference among the results. To get a better understanding of how the benchmarks were run, the following description is provided:

Each benchmark uses fixed data types, bytes, for reproducibility of results (C++: int8_t, Java: byte)

- **Fixed Size Allocation:** Executed with `100,000,000` iterations and an allocation of arrays of `100`-long bytes per allocation.
- **Complex Object Allocation:** Executed with `50,000,000` iterations and `1,000` elements per complex object (the design of the complex object has been described above). Objects were deleted once every 10,000 iterations.
- **Variable Size Allocation:** Executed once using allocation sizes in bytes, loaded from an external file containing `1,000,000` entries, where each entry specifies the length of an byte array to be allocated.
- **Concurrent Multithreaded Allocation Performance:** Executed once using allocation sizes in bytes, loaded from an external file(`200,000` entries), where each entry specifies the length of an byte array to be allocated, running across `40` threads concurrently.
- **Fragmentation Test (Random Allocation and Release):** Executed once using `10,000,000` allocation sizes in byte arrays and corresponding free patterns, both loaded from external files. The probability of freeing memory in the configuration file was 50 to 10000000.
- **Recursive Memory Allocation:** Executed once with a recursion depth of `100,000` and an allocation size of `32,768` bytes per call.

Table 2. Comparison of Benchmark Results

Benchmark	**C++ (ms)**	**Java (ms)**	**% Faster (Java vs C++)**
Fixed Size Allocation	4502.77	5536.21	-22.95%
Complex Object Allocation	8162.08	18685.64	-128.93%
Variable Size Allocation	822.81	145.69	82.29%
Concurrent Multithreaded Allocation Performance	7088.08	5099.10	28.06%
Fragmentation Test Random Allocation Release	5738.97	18361.38	-219.94%
Recursive Memory Allocation	13808.80	526.73	96.19%

8 DISCUSSION

The comparison of the benchmarking results provides valuable insights into C++ and Java memory management's performance aspects. We provide further detailed explanations below for each benchmarking class.

8.1 Fixed Size Allocation

For Fixed Size Allocation, time difference is not notable, but there is anyway. Most likely it is because in some point GC is invoked in Java, which will start to move these objects to old generation, and that slows it down. Otherwise, the allocation itself will most likely remain the same, because at small values it was not noticeable.

8.2 Complex Object Allocation

In the case of Complex Object Allocation, we see a huge gap in time. And we should understand that now the benchmark removes objects once in 10000 iterations, before that the benchmark was simply not executed for any time. This is due to the fact that we explicitly call the garbage collector in Java. The garbage collector may not be called, but it is required to take this instruction into account. As a result, the garbage collector starts to be called too often and all threads are frozen for safe garbage collection, as a result it slows down Java catastrophically.

8.3 Variable Size Allocation

For Variable Size Allocation, the time difference is not as noticeable, the values here are generally very small. We don't measure deallocation on every iteration here, so Java doesn't call the garbage collector, so it doesn't freeze the thread and doesn't move objects to Old generation too often. This speeds up the process because deallocation to New generation in Java is almost instantaneous, unlike the more complex allocation process in C++. Most likely, this is what creates such a time gap.

8.4 Concurrent Multithreaded Allocation Performance

For Concurrent Multithreaded Allocation Performance, or the multithreaded version of the benchmark above, things are a bit more interesting. Here the gap is already very noticeable, 2 seconds. This is due to the fact that Java uses thread-isolated memory areas, while C++ has shared access to a single memory, which can lead to excessive fragmentation. This slows down memory handling processes and as you can see, significantly.

8.5 Fragmentation Test Random Allocation Release

Tests on memory fragmentation with simulation of different allocation sizes and random release patterns also showed a significant difference and as in Complex Object Allocation this is due to the fact that frequent arrival of the garbage collector slows down the program catastrophically. It has to freeze threads, rearrange objects, etc. which is terribly time consuming. There are no such problems in C++ because we just delete a block once and that's all. We get fragmentation but we don't waste time on rearranging objects and freezing threads. Perhaps it is not so efficient in terms of memory at the moment, but we are only interested in speed.

8.6 Recursive Memory Allocation

In the case of recursive memory allocation, there was a problem with the fact that the C++ stack is much smaller and C++ does not optimize recursive operations in any way. At this time Java inlines recursion and optimizes through JIT, which speeds up both memory and overall runtime. It is quite likely that in this case Java just managed to optimize recursion and memory allocation in it, while C++ failed to cope with this task and got a colossal gap.

8.7 Overall Observations

The bottom line is that Java is more optimized for working with short lived objects. The overhead of moving objects to long-term memory and the overall garbage collector work is too high. Whereas for C++ it is critical to avoid fragmentation as it slows down the memory manager significantly, but we don't have the overhead of deleting and rearranging objects. The fragmentation problem will always arise in a multithreaded environment due to the lack of thread-isolated memory space, hence Java will definitely run faster in multithreading if you don't call the garbage collector too often.

These findings suggest that while Java gains an advantage on the fronts of memory management ease and multithreaded performance, C++ remains more apt for low-level operations where low overhead and direct control are the priority.

9 CONCLUSION

In this paper, we conducted a comprehensive comparative analysis of memory management performance of C++ and Java using a suite of synthetic benchmarks. The tests examined a range of memory allocation scenarios including fixed-size allocation, intricate object allocation, concurrent block size allocation, multithreaded concurrent allocation, memory fragmentation, recursive allocation, and random memory access. Findings of this study can be summarized as follows:

- **Fixed-Size Memory Allocation:** Java shows a slight time loss due to shifting objects between new and old object generation partitions, but is otherwise quite fast.
- **Complex Object Allocation:** Because of frequent invocation of garbage collector Java can't work fast enough, because garbage collector blocks threads and constantly transfers objects between memory generations, which slows down the whole work terribly much, so you should avoid frequent invocation of garbage collector.
- **Variable Block Size Allocation:** In situations where we need to allocate many objects of different sizes, Java does it much faster because in New Generation allocation is very fast without complicated mechanisms and due to the lack of frequent garbage collector calls we don't have to freeze threads or rearrange objects, therefore it will work faster than in C++ because there are no complicated memory allocation mechanisms.
- **Concurrent Multithreaded Allocation Performance:** In the multithreaded scenario of allocating blocks of different sizes, Java also shows a good advantage, primarily because of thread-isolated memory. It also helps us again that objects are allocated very quickly in new generation and the garbage collector is not constantly called. C++ slows down due to the fact that it has a shared memory partition, which is highly fragmented due to the allocation of blocks of different sizes in several threads at once, which slows down the memory manager a lot.
- **Fragmentation Test Random Allocation Release:** In the case of this benchmark, things are a bit more complicated. Fragmentation itself, as benchmarks above show, is not terrible for Java, but everything is spoiled by frequent garbage collector calls. For C++ fragmentation is terrible on the contrary, but the slowdown is not so significant, we just use a little more memory at the moment. In Java, huge overheads on the garbage collector slow down the program to catastrophic values. Hence fragmentation is not a problem for Java, but if we want to immediately delete unused blocks of memory, it just slows down a Java application a lot.
- **Recursive Memory Allocation:** Recursive memory allocation also showed that Java has many times better optimization in some places and that if the garbage collector is called correctly, not often Java is able to significantly outperform C++ in terms of memory usage time due to internal optimizations.

In general, these observations suggest that memory speed in Java depends primarily on the competent invocation of the garbage collector. It is impossible to control its calls explicitly, so in real applications the speed of work can be both higher and lower than in C++, but manual memory management in C++ can guarantee efficient memory management, while in Java we cannot do so. However, there are common scenarios where Java is guaranteed to work faster, for example, multithreading, due to another memory.

When is memory management in Java faster? Based on all this information, we can confidently say that Java will be faster when working with objects with a small lifetime and when we have a multi-threaded environment. In such scenarios, Java has special optimization. In C++, regardless of whether we work with some type of objects or utilize them in some manner—the memory manager always behaves the same.

Why is one memory management approach faster than the other? Most significantly, Java's memory manager is slower than that of C++ due to the fact that it entails complex logic. It does not merely delete objects. It must first freeze the threads of the program in order to safely scan objects, move them between regions of memory, or delete them. Whereas C++ just allocates and deallocates objects, occasionally optimizing the operations to avoid fragmentation, this requires far less overhead.

Future research will extend this analysis by incorporating real use cases and more complex memory allocation scenarios, i.e., with objects featuring complex relationships and multiple fields. Additional experimentation with the effects of compiler optimizations (e.g., `-O3` and other options in `CMakeLists.txt` for C++ and `build.gradle.kts` for Java) is also warranted, as well as techniques for disabling overly aggressive optimizations that may invalidate the measurement of "useless" memory operations.

In conclusion, the selection of a memory management strategy needs to be strongly linked to the requirements of the application. The compromise between ease of use, automatic garbage collection, and the aspiration for low-level control remains a significant concern for programmers who operate in environments that call for both performance and reliability.

10 FUTURE WORK

Potential avenues for further research include:

- Expanding the benchmark suite to incorporate real-world applications.
- Analyzing the impact of various compiler optimizations and JVM tuning parameters.
- Developing a unified testing framework that can be used across different programming languages.
- Designing a benchmark specifically focused on testing memory allocation for complex objects with intricate relationships. This would involve objects with numerous fields, nested structures, and references to other objects, closely mimicking real-world application scenarios such as object graphs in enterprise software or large-scale data structures used in scientific computing.

REFERENCES

¸alves, Carlos Daniel Oliveira Gonc (2021). “APerformance Comparison of Modern Garbage Collectors for Big Data Environments”. In: *Examination Committee* 5.3, pp. 51–76.

Balosin, Ionut (2019). “JVM Garbage Collectors Benchmarks Report 19.12”. In: *ionutbalosin*.

Daniel Haggander, Per Lidkn et al. (2001). “A Method for Automatic Optimization of Dynamic Memory Management in C++”. In: *Blekinge Institute of Technology* 3, pp. 491–494.

Goetz, Brian (2024). *Java Memory Model*. Accessed: 2024-04-02. URL: https://shipilev.net/blog/2014/jmm-pragmatics/.

Islam, Md. Jahidul (2023). “Performance Analysis of Modern Garbage Collectors using Big Data Benchmarks in the JDK 20 Environment”. In: *5th International Conference on Sustainable Technologies for Industry* 1.3, pp. 3–6.