Object-Oriented Performance Tuning: Comparing JVM, CPython, and C++

Chetan Badiger, Prajwal Kiran Naik
University of Colorado, Boulder

Abstract

This work aims to compare how the internal object models of different languages influence the speed of the application, memory and the responsiveness to user input. This paper aims to examine three popular environments – JVM, CPython, and native C++ - and analyze how they represent objects in memory. This paper will explore aspects such as virtual tables and garbage collection versus memory management. Prior work highlights the substantial cost of virtual dispatch in C++ (Driesen et, al. 1995), object-centric memory inefficiencies in managed languages, and orders-of-magnitude slowdowns in Python relative to C++ due to interpreter overhead (Saqib et. al. 2021). By comparing these runtimes, the paper emphasizes how OO principles such as encapsulation and polymorphism are implemented differently across different compilers and how these differences affect CPU cycles, cache locality, and runtime optimizations.

Motivation

Performance is a critical factor in modern software, ranging from high frequency trading platforms to healthcare systems. Although Object-Oriented programming improves maintainability, their advantages can incur additional compilation and memory overheads that can vary by language and implementations. By understanding how OO concepts behave in JVM, CPython and C++ applications, developers can make educated decision about the selection of language for their applications (Rakhimov 2024). This topic is significant to the wider software engineering community as it connects language theory and systems level optimizations. This potentially enables better design decisions for both high and low level performance engineering.

Introduction

Object-oriented Programming principles represent one of the most sought-after topics of discussion in industrial software development (Larman, 2005). The popular principles of OOP, namely, encapsulation, inheritance, polymorphism and abstraction empower modularity, maintainability and scalability in large-scale codebases (Gamma et al., 1994). Nevertheless, the practical efficiency of OOP depends on both language design and compiler implementation. Different interpretations of OOP features by different compilers affect execution speeds, memory consumption and binary code efficiency.

This paper focuses on the compiler-level implementation of key OOP principles across three popular programming languages – Java, C++ and Python. These languages use three distinct compilation models, native bytecode Just-In-Time compilation, native compilation and interpreted execution respectively. We aim to analyze how these compilation designs influence performance and resource utilization. This research aims to clarify trade-offs that development teams face when design large-scale, high-performance systems.

Literature Review

Compiler designs and their performance implications have been a widely studied topic. Previous research describe different compiler stages – lexical analysis, parsing, optimization and code generation – while emphasizing that OOP designs varies widely between different compilers (Aho et el., 2006). The Java Virtual Machine (JVM) has been research extensively, highlighting how its adaptive optimization and dynamic class loading allow efficient execution of polymorphic calls at runtime. Effective optimizations like inline caching and dynamic de-virtualization enable this in JVM. In contrast, the GCC and Clang compilers utilize static dispatch and vtable-based dynamic dispatch, enabling heavy optimization at compile time (Stroustrup, 2013).

The Python interpreter utilizes a distinct approach from the other two languages. It relies on reference counting, dynamic typing and interpreted execution, all of which introduce significant overhead in method resolution (Van Rossum & Drake, 2009).

Prior research which compares multi-language performance (Prechelt, 2000) finds that although compiled languages perform better than interpreted ones, modern JIT optimization techniques narrow down this difference. This paper aims to explicitly attribute this difference to compiler handling of OOP features.

Methodology

This study focuses on four primary OOP aspects. First, valuating how access control mechanisms influence performance and memory usage (encapsulation). Second, measuring how dynamic dispatch impacts execution speed and inlining potential (Polymorphism). Third, Assessing the influence of optimization flags on OOP code efficiency (compiler optimization).

Experimental Setup

This study's experiments are conducted on a ARM64 machine with 16 GB RAM running the MacOS Sequoia 15.6. The benchmark codes include concise, reproduceable and semantically equivalent OOP code snippets across three languages:

- C++ - GCC 17.0.0
- Java - OpenJDK 24.0.2
- Python - 3.13.1 (Clang 16.0.0)

Encapsulation Benchmark

To test the performance (execution time and memory usage) for encapsulation, the code snippet in Listing 1 was used for C++. Similar analogues were used for Python and Java. Care has been taken to ensure that the snippets are as close to each other as possible.
To record the execution times, the following libraries were used:

- C++ - chrono.
- Python - time
- Java - System

Listing 1: C++ Encapsulation `encapsulation.cpp`

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;
class Account {
private:
  double balance;
public:
  Account(double b) : balance(b) {}
  double getBalance() const { return balance; }
  void setBalance(double b) { balance = b; }
};
int main() {
int N = 10'000'000;
  auto start = high_resolution_clock::now();
  for (int i = 0; i < N; i++) {
    Account newAccount(100);
      newAccount.setBalance(200);
       double newBalance = newAccount.getBalance();
  }
  auto end = high_resolution_clock::now();
  auto duration = duration_cast<nanoseconds>(end - start);
  cout << "Execution time: " << duration.count() << endl;
  return 0;
}
```

Polymorphism Benchmark

To test the performance for polymorphism, the code snippet in Listing 2 was used for C++. As in the case of encapsulation, similar analogues were used for Python and Java as well.

Listing 2: C++ Encapsulation `encapsulation.cpp`

```
#include <iostream>
#include <chrono>
using namespace std;
class Shape {
public:
  virtual double area() const { return 0.0; }
};
class Circle : public Shape {
public:
   double area() const override { return 3.14 * 5 * 5; }
};
int main() {
   int N = 10'000'000;
   auto start = chrono::high_resolution_clock::now();
   for (int i = 0; i < N; i++) {
       Shape* shape = new Circle();
       shape -> area();
       delete shape;
   }
   auto end = chrono::high_resolution_clock::now();
   cout << chrono::duration<double>(end - start).count() << endl;
}
```

Compiler Optimization Benchmark

To test the performance for compiler optimizations, the code snippet in Listing 3 was used for C++. As in the case of encapsulation and polymorphism, similar analogues were used for Python and Java as well.

To test C++ compiler optimizations, three optimization flags were used –

- -O0 - no inlining, no dead-code elimination
- -O2 - compiler inlines & optimizes loop
- -O3 - aggressive inlining + improved vtable prediction

To test the Java Compiler Optimizations, we used two java compiler modes –

- -Xint - Only interpreter mode
- Mixed mode (default JIT) – Normal execution

The Python compiler does not expose any compiler optimizations to the developer. Hence, the Python analogue of Lising 3 was run on default compiler and it's timing was recorded.

Listing 2: C++ Optimization Compiler `main.cpp`

```
#include <iostream>
#include <chrono>
using namespace std;

class Shape {
public:
    virtual double area() const { return 0.0; }
};

class Circle : public Shape {
public:
    double area() const override { return 3.14 * 10 * 10; }
};

int main() {
    int N = 20'000'000;

    auto start = chrono::high_resolution_clock::now();
    double total = 0.0;

    for (int i = 0; i < N; i++) {
        Shape* s = new Circle();
        total += s->area();
        delete s;
    }

    auto end = chrono::high_resolution_clock::now();
    cout << chrono::duration<double>(end - start).count() << endl;
    return 0;
}
```

Results

Table1 shows the comparison between the execution speeds for different programing languages while implementing the encapsulation and polymorphism constructs.

	C++ (s)	Java (s)	Python (s)
Encapsulation	8.3e-08	0.004987	1.561383542
Polymorphism	8.3e-08	0.002268958	0.055025

Table 1: Execution time of each programing language in seconds

Table 2 shows the comparison between the memory utilization of the different programing languages.

	C++	Java	Python
Encapsulation	378159104	64	360
Polymorphism	378126336	10544	72

Table 2: Memory Utilization of Each Programming Language in Bytes

Table 3 shows the comparison of execution times for C++ with different compiler optimization flags.

Optimiza-tion Flags	-O0	-O2	-O3
Execution Times (s)	0.435	6.93e-08	6.9e-08

Table 3: Memory Utilization of Each Programming Language in Bytes

Table 4 shows the execution time comparision for Java code samples.

Optimization Flags	-Xint (Only in-terpreter)	With JIT
Excution Times (s)	1.166	0.0034

Table 4: Memory Utilization of Each Programming Language in Bytes

Since the Python compiler does not expose compiler flags, it's execution time is 2.7788s.

Conclusion

From Table 1, it is evident that C++ consistently outperforms Java and Python in execution time, with operations completing in the order of nanoseconds. Java performs moderately, benefiting from Just-In-Time (JIT) compilation, whereas Python exhibits substantially slower performance due to its interpreted nature. Notably, polymorphism shows a smaller performance gap between Java and Python, suggesting that dynamic dispatch overhead in Python is more pronounced in simple operations like encapsulation.

Table 2 reveals a contrasting trend in memory usage. C++ requires significantly more memory compared to Java and Python in absolute terms, likely due to the way memory is allocated and reported in the benchmarking environment. Java demonstrates efficient memory management, while Python shows moderate memory usage depending on the operation.

The influence of compiler optimizations further supports the differences between these languages. Table 3 shows that C++ has dramatic benefits from aggressive optimization flags: while unoptimized code (-O0) executes in approximately 0.435 s, optimized binaries using -O2 or -O3 improve performance by several orders of magnitude, reducing execution time to roughly 7×10^{-8} seconds. This demonstrates the huge impact of static compilation and ahead-of-time optimization on OOP constructs, particularly dynamic dispatch and object creation.

Java follows similar trends as shown in Table 4, running Java in interpreter-only mode (-Xint) produces significantly slower execution times (1.166 s), whereas enabling the default tiered JIT results in substantial speedups (0.0034 s). Java does not expose different flags it will rather perform optimizations implicitly such as inlining, escape analysis and devirtualization at runtime.

Python on other hand shows minimal opportunity for compiler-level performance improvements. Since Cpython does not provide optimization flags or JIT compilation, its execution time remains high comparatively. This emphasizes the inherent cost of Python's dynamic object model, reference-counting memory management, and interpreter overhead when executing OOP-heavy workloads.

Overall, the results across all the four highlight the fundamental trade-offs between native, managed and interpreted execution models. C++ provides significant performance when done with compiler optimization but requires greater memory overhead and manual memory management. Java offers ease of use and expressive power at the cost of execution speed. Software Engineers should consider these distinctions when selecting language for performance-critical, object-oriented applications, especially in domains where optimization flags, runtime compilation strategies, and object layout efficiency significantly affect real-world performance.

References

Larman, C. 2002. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Upper Saddle River, NJ: Prentice Hall PTR.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Aho, A. V.; Lam, M. S.; Sethi, R.; and Ullman, J. D. 2006. Compilers: Principles, Techniques, and Tools. 2nd ed. Boston: Addison-Wesley.

Stroustrup, B. 2013. The C++ Programming Language. 4th ed. Upper Saddle River, NJ: Addison-Wesley.

Van Rossum, G., and Drake, F. L. 2009. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

Prechelt, L. 2000. *An Empirical Comparison of Seven Programming Languages*. *Computer* 33(10): 23–29.

Rakhimov, R. (2024). Object-oriented programming languages comparison: Java, Python, and C++.

Karel Driesen and Urs Hölzle. 1996. The direct cost of virtual function calls in C++. SIGPLAN Not. 31, 10 (Oct. 1996), 306–323. https://doi.org/10.1145/236338.236369

Ali, Saqib & Qayyum, Sammar. (2021). A Pragmatic Comparison of Four Different Programming Languages. 10.14293/S2199-1006.1.SOR-.PP5RV1O.v1.

Pape, Tobias & Bolz, Carl Friedrich & Hirschfeld, Robert. (2017). Adaptive just-in-time value class optimization for lowering memory consumption and improving execution time performance. *Science of Computer Programming*

Zhang, Mengchi & Alawneh, Ahmad & Rogers, Timothy G. (2021). Characterizing Massively Parallel Polymorphism. Proceedings of the 2021 IEEE International Symposium on *Performance Analysis of Systems and Software (ISPASS)*, 205–216

Gossen, Frederik & Jasper, Marc & Murtovi, Alnis & Steffen, Bernhard. (2019). Aggressive Aggregation: A New Paradigm for Program Optimization. arXiv preprint arXiv:1912.11281. https://arxiv.org/abs/1912.11281

Shuf, Yefim & Gupta, Manish & Bordawekar, Rajesh & Singh, Jaswinder. (2002). Exploiting Prolific Types for Memory Management and Optimizations. ACM SIGPLAN Notices. 10.1145/503272.503300.

Springer, Matthias & Masuhara, Hidehiko. (2018). DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access. arXiv preprint arXiv:1810.11765. https://arxiv.org/abs/1810.11765

Li, Bolun & Xu, Hao & Zhao, Qidong & Su, Pengfei & Chabbi, Milind & Jiao, Shuyin & Liu, Xu. (2022). OJXPerf: Featherlight Object Replica Detection for Java Programs. arXiv preprint arXiv:2203.12712. https://arxiv.org/abs/2203.12712