Quantification of the Abstraction Penalty

Caleb Bettcher[1], **Kerem Gurkan**[1],

[1]University of Colorado Boulder
caleb.bettcher@colorado.edu, kerem.gurkan@colorado.edu

Abstract

Modern software design patterns heavily rely on inheritance–a fundamental mechanism through which a child class acquires and extends functionality defined by a parent. While inheritance has become widespread due to its support for code reuse and modularity, its interaction with compilers raises a less-examined question: does abstraction introduce quantifiable performance costs at runtime? This paper empirically investigates that question by measuring the performance impact of three key mechanisms: static versus dynamic binding, dynamic dispatch behavior, and inheritance depth. Experiments are conducted using Java Microbenchmark Harness (JMH) micro-benchmarks under three JVM optimization configurations: standard JIT compilation, inlining disabled, and fully interpreted mode. We find that under standard JIT the binding penalty is modest (approximately 20% over a static baseline), and chain depth has no measurable effect when intermediate layers perform no work. When intermediate layers actively chain `super()` calls, however, the abstraction penalty scales dramatically with depth: interpreted-mode costs grow approximately linearly at roughly 17 ns per additional inheritance level, reaching a 3,646% overhead at depth 128 compared to the static baseline.

Introduction

Inheritance is one of the cornerstones of object-oriented programming. By allowing child classes to inherit fields and methods from parent classes, it significantly reduces code duplication and enhances maintainability across large codebases. Beyond the structural benefits, inheritance also enables runtime polymorphism through dynamic method dispatch, a mechanism that allows the JVM to resolve method calls at runtime rather than compile time–an integral part of many modern design patterns including the Template and Strategy patterns.

However, this flexibility may not come for free. Abstraction mechanisms, including dynamic dispatch and superclass-typed invocations, require the JVM to perform additional operations at runtime. Crucially, many of these costs are masked in practice by Just-In-Time (JIT) compiler optimizations such as speculative inlining and devirtualization. Whether and when these costs become visible and how dramatically they scale depends on both the structure of the inheritance hierarchy, and the optimized environment in which it executes.

To investigate these questions, we designed and executed a suite of micro-benchmarks targeting three independent variables: binding mechanism (static vs. dynamic), invocation type (superclass-typed vs. concrete-typed), and inheritance depth. Each micro-benchmark is evaluated under either a chain design in which intermediate layers perform no work, or one in which each layer contributes an active `super()` call. Each chain design is also evaluated with three varying JVM configurations that progressively strip away JIT optimizations. The contributions of this paper are as follows:

- A quantification of the abstraction penalty attributable to inheritance-related mechanisms in Java, measured across standard JIT, inlining-disabled, and fully interpreted execution environments
- An empirical demonstration that inheritance depth is only a performance factor when intermediate layers contribute live `super()` dispatch, at which point costs scale dramatically and can exceed 3,600% over a static baseline in interpreted mode
- An interpretation of these findings for real-world applications, including domains where nanosecond-level performance is critical

Background

JIT Compilation

Java source code is compiled into platform-independent bytecode, which the Java Virtual Machine (JVM) then executes. Rather than interpreting every bytecode instruction individually (a slow approach), modern JVMs include a Just-In-Time (JIT) compiler that monitors execution at runtime and compiles frequently executed code paths ("hot spots") directly into native machine code (Suganuma et al. 2005).

Crucially, the JIT has access to runtime information that a conventional ahead-of-time compiler does not. It can observe which concrete types actually appear at a call site, how often a branch is taken, and whether a method body is small enough to replace the call with an inlined copy of its instructions. These profile-guided optimizations including *speculative inlining*, *devirtualization*, and *branch prediction* routinely eliminate overhead that would otherwise be imposed by Java's dynamic dispatch model (Suganuma et al. 2005). This masking effect is central to our experiments: the three

JVM configurations we test (standard JIT, inlining disabled via `-XX:-Inline`, and fully interpreted via `-Xint`) progressively strip away these optimizations, allowing the underlying dispatch costs to emerge.

Static vs. Dynamic Binding

Method binding refers to the process of associating a method call with its corresponding implementation. In Java this can occur either at compile time (static binding) or at runtime (dynamic binding). Static binding applies to private, static, and final methods, where the compiler can unambiguously determine the target method. Because this resolution is done ahead of time, statically bound calls are candidates for inlining by the JIT compiler, potentially eliminating function call overhead entirely.

Dynamic binding, by contrast, applies to virtual methods that may be overridden by subclasses. The JVM resolves these calls at runtime by consulting the virtual method table (vtable), a per-class data structure that maps method signatures to their concrete implementations (Dean, Grove, and Chambers 1995). This lookup introduces overhead that static binding avoids. However, the JIT compiler can sometimes devirtualize dynamic calls if it determines at runtime that only one implementation is ever invoked (Dean, Grove, and Chambers 1995).

Superclass-Typed vs. Concrete-Typed Invocation

Even when a method is repeatedly called on the same object, the declared type of the reference variable can affect how the JVM resolves the call. When a method is invoked through a concrete-typed reference (e.g., `Circle myShape = new Circle()`), the JVM has precise type information available at the call site, which may allow the JIT to apply static binding or aggressive in-lining (Lenovo CA n.d).

When the same call is made through a superclass-typed reference (e.g., `Shape myShape = new Circle()`), the superclass type provides less information and thus the JVM must rely on dynamic dispatch to determine the correct implementation at runtime (Lenovo CA n.d). This distinction is directly tied to the behavior of polymorphic code: accepting a superclass reference is what makes a method polymorphic, but it also prevents certain compiler-level shortcuts. We specifically test whether this declared-type difference introduces a measurable runtime cost independent of the actual object being operated on.

Abstraction Penalty

The abstraction penalty, as used in this paper, refers to the measurable runtime overhead introduced by abstraction mechanisms beyond what is required to execute the underlying logic (Luján, Freeman, and Gurd 2005). It is important to note that the inheritance hierarchy itself–the structural relationship between classes–does not inherently cause performance degradation. Rather the penalty arises from runtime behaviors that inheritance enables: vtable lookups for virtual dispatch, missed in-lining opportunities when call targets cannot be determined statically, and incorrect branch predictions when the JVM's speculative optimizations guess incorrectly about which implementation will be called (Luján, Freeman, and Gurd 2005). Isolating these factors from the functional behavior of the tested code is a primary challenge of our experimental design.

Polymorphism Scalability

The JVM applies different optimization strategies depending on how many distinct concrete types are observed at a given call site (Kaleba et al. 2022). A monomorphic call site is one where only a single concrete type has ever been observed; the JVM can speculatively inline the method, effectively eliminating dispatch overhead entirely. A bimorphic call site involves two observed types, allowing for an efficient two-way branch. A megamorphic call site–one where three or more distinct types appear–triggers a general vtable lookup as the JVM gives up on type-specific speculation (Kaleba et al. 2022).

This scaling behavior has direct implications for polymorphic design patterns. Code that appears clean and flexible at the architectural level may introduce a performance cliff once a call site becomes megamorphic (Bogosavljević 2022). This progression is a known limitation of devirtualization-dependent optimizations and represents a direction for future empirical investigation beyond the monomorphic scope of the present study.

Related Work

Prior work on the performance implications of abstraction and inheritance in object-oriented languages spans several distinct threads including the characterization of abstraction penalties, the design and evaluation of JIT optimization techniques, the empirical measurement of polymorphic call-site behavior, and the structural analysis of inheritance hierarchies. We worked carefully to connect these threads while targeting a gap none of them fully addresses.

Abstraction Penalties

The most directly related line of work concerns the runtime costs of high-level abstractions in Java. Luján et al. (Luján, Freeman, and Gurd 2005) examined the conditions under which abstraction penalties can be eliminated in OOLALA, an object-oriented linear algebra library implemented in Java. Their findings showed that high-abstraction implementations were not performance-competitive with low-abstraction baselines, and they characterized the specific storage formats and matrix properties under which the gap could be closed. While their study is grounded in a domain-specific library rather than general dispatch mechanisms, it establishes the foundational framing we adopt: abstraction penalties are conditional and depend on whether the runtime can reduce high-level constructs to their low-level equivalents.

Müller (Müller 2000) investigated abstraction benchmarks in C++ on parallel vector architectures, finding that advanced techniques such as expression templates could bring C++ performance in line with C and Fortran. This

work demonstrates that the abstraction penalty is not unique to Java and is instead a broader consequence of object-oriented indirection–one that depends heavily on what the compiler can see and eliminate. Bogosavljević (Bogosavljević 2022) provides a more recent and practical characterization of the same problem in the context of polymorphic class processing, showing that the layout of objects in memory and the predictability of call targets have substantial effects on throughput. Together, these works establish that abstraction penalties are real, measurable, and compiler-dependent across languages–motivating our JVM-specific investigation.

JIT Optimization and Dynamic Dispatch

Suganuma et al. (Suganuma et al. 2005) provide the most directly relevant treatment of JIT design in the Java context. Their work describes a production-level JIT compiler employing a three-tier optimization framework with profile-directed method inlining and code specialization. Critically, they demonstrate that the JIT's ability to inline virtual calls is contingent on call-site stability and method body size–the same budget constraints our SuperChaining results expose empirically. Their mixed-mode interpreter and tiered compiler architecture is the foundation upon which HotSpot is built, and our three JVM configurations directly correspond to different tiers of the optimization pipeline they describe.

Dean et al. (Dean, Grove, and Chambers 1995) address the problem of eliminating virtual dispatch overhead through static class hierarchy analysis, showing that even without runtime profiling, a compiler can devirtualize a significant fraction of virtual calls by reasoning about which concrete types are reachable. Their analysis is particularly relevant to our Return0 results: the flat depth curve we observe under all three JVM configurations is consistent with the JIT applying a form of the devirtualization Dean et al. describe, collapsing the hierarchy to a single stable call target regardless of declared depth.

Polymorphism and Call-Site Behavior

Choi and Tempero (Choi and Tempero 2007) developed a dynamic metric for polymorphism based on observed runtime behavior rather than static code structure, finding through case studies that the dynamic profile of polymorphic dispatch differs substantially from what static analysis would predict. This motivates our use of JMH and runtime measurement over static analysis: the true dispatch cost of an inheritance hierarchy cannot be determined from the class structure alone.

Kaleba et al. (Kaleba et al. 2022), studying call-site behavior in Ruby applications, found that the vast majority of call sites are monomorphic in practice and that lookup caches and method splitting are consequently highly effective. While their work targets a different language and runtime, the fact that most real-world call sites are monomorphic supports the ecological validity of our monomorphic experimental design as a baseline, while also reinforcing that megamorphic scenarios represent a distinct regime warranting separate study.

Inheritance Structure and Complexity

A separate body of work examines inheritance from a structural and complexity perspective rather than a performance one. Singh (Singh 1995) surveyed single and multiple inheritance mechanisms across languages, characterizing the tradeoffs between expressiveness and implementation complexity. Dinesh (Dinesh 1992) extended the theoretical treatment of inheritance by generalizing the notion of delegation and proposing adoption as a mechanism for resolving type conflicts in deep hierarchies. Misra et al. (Misra, Akman, and Koyuncu 2011) proposed a cognitive complexity metric for inheritance, finding that hierarchy depth correlates with maintenance burden–a structural concern that complements our runtime findings. Crisan (Crisan 2020) examined inheritance versus composition in complex systems architecture, and Kumar (Kumar 2022) provides a practical treatment of the same design tradeoff.

While none of these works measure runtime dispatch costs directly, they collectively establish that inheritance depth is a recognized design variable with consequences across multiple dimensions–structural, cognitive, and as we demonstrate, performance-related under the right conditions. Our work is distinguished from this body of literature by its exclusive focus on the runtime dispatch mechanisms that deep hierarchies enable, rather than on the hierarchies themselves as design artifacts.

Methodology

Our experimental methodology is designed to isolate the runtime cost of specific inheritance-related mechanisms by constructing functionally equivalent code-bases that differ only in the abstraction mechanism under examination.

Identification of Variables

We measured the impact of binding mechanisms (static vs. dynamic) and inheritance chain levels (1-128 depth). All chain instances are held behind superclass-typed references, making invocation type a controlled constant across all benchmarks rather than an independent variable. All timing is reported as mean nanoseconds per operation with standard deviations aggregated over repeated trials. Two distinct chain implementations are evaluated, described below.

Code-base Structure

The codebase is organized around three components Listing 1: the abstract shape class defines the contract; "`CircleStatic`" provides a non-polymorphic static baseline; and `Circle` (depth 1) is the concrete dynamic dispatch counterpart, stored behind a `Shape` reference to force a vtable lookup.

To evaluate the effect of inheritance depth, we constructed two families of chain classes at depths 2, 4, 8, 16, 32, 64, and 128, differing in how intermediate layers participate in computation.

- **Return chains** (Listing 2, top): each intermediate layer overrides `area()` and returns 0 directly without invoking `super()`. Only the bottom-most concrete class performs the actual computation. This design ensures that

Listing 1: Core class hierarchy: abstract base, static utility, and depth-1 concrete class.

```
// Abstract base class
public abstract class Shape {
  public abstract double area();
}
// Static utility -- no dispatch
    overhead
public final class CircleStatic {
  public static double area(double r) {
    return Math.PI * r * r;
  }
}
// Depth-1 concrete class (held as Shape
    ref)
public class Circle extends Shape {
  private final double radius;
  public Circle(double r) { this.radius
      = r; }
  @Override
  public double area() {
    return Math.PI * radius * radius;
  }
}
```

the vtable always resolves to a single call target regardless of chain depth, so the JIT sees an optimizable stable monomorphic call site.

- **SuperChaining chains** (Listing 2, bottom): each intermediate layer calls `super.area()` and adds its own contribution, creating a genuine chain of virtual method invocations that traverses the full hierarchy on every call. This design forces each layer to participate in dispatch, making the abstraction cost of depth observable under the aggressively optimizing JIT compiler.

The Github Repository can be found at the following url: https://github.com/CBetch/inheritance-cost-benchmark.

JMH Benchmark Harness

All benchmarks are structured using the Java Microbenchmark Harness (JMH) as shown in Listing 3. A `Blackhole` sink is used at every call site to prevent dead-code elimination of computed results.

Environmental Configurations

All benchmarks were executed using JMH 1.37 under Java 21 on an Amazon EC2 instance. Three iterations were used for warmup, followed by 20 measurement iterations per fork. Each complete suite was executed 5 times to assess result stability across runs; reported means are averaged across those 5 runs. Three JVM configurations were tested:

1. **Standard JIT**: default HotSpot C2 with all optimizations enabled
2. **No-Inline**: JIT enabled but inlining suppressed via `-XX:-Inline`
3. **No-JIT**: fully interpreted mode via `-Xint`, disabling all JIT compilation.

Listing 2: Return0 (top) vs. SuperChaining (bottom) with depth-8 chain implementations. Return0 resolves to a single call target; SuperChaining traverses every level.

```
// Return0:
// depth-8, each layer returns 0
class Chain8L1 extends Shape {
  @Override public double area() {
      return 0; }
}
// L2..L7 also override: return 0
//(No super call)
public class Chain8 extends Chain8L7 {
  @Override
  public double area() {
    return Math.PI * radius * radius; //
        only real work
  }
}

// SuperChaining:
// depth-8, each layer chains
class SC8L1 extends Shape {
  @Override public double area() {
      return 0; }
}
class SC8L2 extends SC8L1 {
  @Override public double area() {
    return super.area() + 0;
  }
}
// L3..L7 similarly call super.area() +
    0
public class SC_Chain8 extends SC8L7 {
  @Override
  public double area() {
    return super.area() + Math.PI *
        radius * radius;
  }
}
```

Results

Results are presented in three tables. Table 1 compares static versus dynamic dispatch at a single inheritance level (depth 1) across all three JVM configurations, providing a baseline measure of the binding penalty independent of depth. Tables 2 and 3 then report inheritance depth results for the Return0 and SuperChaining designs respectively. All values represent means averaged across 5 independent runs; the `staticCall` benchmark serves as the zero-overhead baseline for computing overhead percentages. The Return0 and SuperChaining designs were benchmarked in separate execution runs; as a result, the `staticCall` baseline differs slightly between Table 1 and Table 3 (0.556 ns versus 0.627 ns under standard JIT), reflecting normal run-to-run variance on a shared cloud instance. Percentage comparisons within each table are computed against that table's own baseline and remain internally consistent.

Listing 3: JMH benchmark harness (abbreviated). Three fork variants are defined (not shown): standard JIT (`jvmArgsPrepend = {}`), inlining disabled (`-XX:-Inline`), and interpreter-only (`-Xint`).

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Thread)
@Warmup(iterations = 3, time = 1,
    timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 20, time = 1,
    timeUnit = TimeUnit.SECONDS)
@Fork(
    value = 1,
    jvmArgsPrepend = {}
)
public class DispatchBenchmark {
    private static final double RADIUS =
        5.0;
    private Shape mono = new Circle(
        RADIUS);
    private Shape chain8 = new Chain8(
        RADIUS);

    @Benchmark
    public void staticCall(Blackhole bh)
        {
        bh.consume(CircleStatic.area(
            RADIUS));
    }

    @Benchmark
    public void dispatch1Layer_mono(
        Blackhole bh) {
        bh.consume(mono.area());
    }

    @Benchmark
    public void chainDepth8(Blackhole bh
        ) {
        bh.consume(chain8.area());
    }
    // chainDepth2, 4, 16, 32, 64, 128
    // are of same pattern
}
```

Static vs. Dynamic Binding

Table 1 shows that under standard JIT the binding penalty is modest: dynamic dispatch costs approximately 0.11 ns more than a static call, a 19.8% overhead. When inlining is suppressed the overhead drops in relative terms (4.1%), reflecting that the static call also loses its primary optimization advantage and both costs converge near the raw method-call floor of approximately 2.8–2.9 ns. Under the interpreter both conditions are substantially more expensive in absolute terms, with the dynamic call incurring a 10.6% penalty—roughly 5.7 ns extra per operation.

Inheritance Depth: Return0 Design

Table 2 presents the Return0 results under standard JIT across all chain depths. The most striking feature is the flatness of the depth curve: every chain depth from 2 to 128 measures within approximately 0.1 ns of the depth-1 dynamic call (0.660–0.691 ns), and the overhead relative to the static baseline remains stable at 18.7–24.3%. Under No-Inline and No-JIT configurations (not shown in the table for brevity), the same flat pattern holds: No-Inline values cluster tightly around 2.84 ns for all depths ($\pm$0.08 ns), and No-JIT values cluster around 57–59 ns with no systematic depth trend.

Table 1: Static vs. Dynamic Dispatch at Depth 1 (Return0, mean of 5 runs)

JVM Config	**Static (ns)**	**Dynamic (ns)**	**Δ (ns)**	**Δ (%)**
Standard JIT	0.556	0.666	+0.110	+19.8
No-Inline	2.805	2.921	+0.116	+4.1
No-JIT	53.566	59.239	+5.673	+10.6

Table 2: Inheritance Depth with Return0 Design under Standard JIT

Depth	**Mean (ns/op)**	**Δ vs. Static (ns)**	**Δ (%)**
Static (baseline)	0.556	—	—
1 (mono)	0.666	+0.110	+19.8
2	0.660	+0.104	+18.7
4	0.660	+0.104	+18.7
8	0.691	+0.135	+24.3
16	0.662	+0.106	+19.0
32	0.662	+0.106	+19.0
64	0.662	+0.106	+19.1
128	0.667	+0.111	+20.0

Inheritance Depth: SuperChaining Design

Table 3 presents the SuperChaining results. In sharp contrast to the Return0 design, depth has a dramatic effect on runtime cost when each layer actively calls `super.area()`.

Under standard JIT, costs at depths 1–8 remain comparable (0.759–0.910 ns), suggesting the JIT successfully inlines the short chains. At depth 16 costs jump to 3.575 ns (+470%), indicating the JIT's inlining budget is saturated around this threshold. The cost does not increase uniformly beyond that point—depth 32 is slightly lower than depth 16 (3.328 ns), while depth 64 jumps to 11.839 ns and depth 128 reaches 37.230 ns (+5,840%)—consistent with the JIT intermittently finding partial inlining solutions for certain depth values.

Under No-Inline, the cost scaling is cleaner and near-linear: each doubling of depth approximately doubles the cost, reflecting the raw expense of un-inlined virtual dispatch at every `super()` call site. Depth 128 reaches 278.262 ns (+8,607% relative to the static baseline).

Under No-JIT (fully interpreted), scaling is almost perfectly linear with depth. From depth 1 to depth 128 the cost grows from 67.473 ns to 2342.081 ns, an increase of approximately 17 ns per additional inheritance level—consistent with the interpreter's fixed per-call overhead for a virtual dispatch.

Table 3: Inheritance Depth with SuperChaining Design under all JVM configs (mean over runs provided as ns/op)

Depth	Standard JIT	No-Inline	No-JIT
Static (baseline)	0.627	3.196	62.516
1 (mono)	0.759	3.228	67.473
2	0.904	4.737	81.325
4	0.910	6.265	108.581
8	0.905	8.733	167.867
16	3.575	15.271	310.974
32	3.328	42.283	570.251
64	11.839	79.675	1116.300
128	37.230	278.262	2342.081

Discussion

The interpretation of our results centers on two themes: first, the conditions under which the JIT eliminates the abstraction penalty entirely; second, the mechanism by which depth produces dramatic overhead when the JIT's optimizations are either suppressed or overwhelmed.

JIT masking in the Return0 design

The flat depth curve in Table 2 demonstrates that the JIT's speculative inlining completely eliminates dispatch overhead when each call site has a single, stable concrete type (monomorphic) and the method body is simple enough to fit within the inlining budget. Even at depth 128, only the bottom-most `area()` implementation ever executes non-trivially; the JIT observes this, devirtualizes the call, and effectively reduces the inheritance hierarchy to a single inlined computation. The 20% overhead that persists at depth 1 over the static baseline is therefore not a depth effect—it is the minimum cost of a virtual call invocation in this environment, which the JIT reduces but does not fully eliminate for a `Shape`-typed reference.

The `super()` chain exhausts the inlining budget

The SuperChaining data reveals the true mechanism behind inheritance depth costs. When each layer contributes a real `super.area()` call, the JIT must inline an entire chain of method frames to eliminate dispatch overhead. For depths 1–8 the chain fits within the JIT's inlining budget, producing costs (0.759–0.910 ns) only slightly higher than depth 1. Around depth 16, the budget is saturated, and the cost jumps by nearly 5×. The non-monotone behavior between depths 16 and 32 (where depth 32 is slightly cheaper) suggests the JIT is making heuristic decisions—possibly re-optimizing with a different inlining strategy—that produce unpredictable cost cliffs rather than smooth scaling.

Linear scaling under interpretation confirms per-call cost

The No-JIT results for SuperChaining provide the cleanest signal: approximately 17 ns is added per additional inheritance level, confirming that each `super.area()` call traverses a full virtual dispatch in the interpreter.

Practical implications

Under standard JVM conditions, developers using shallow inheritance hierarchies (1–8 levels) with monomorphic call sites incur at most a 44% overhead versus a static baseline—but in absolute terms this is well under 1 ns per call, which is negligible for most applications. The practical warning lies in two scenarios: (1) very deep `super()` chains that overflow the inlining budget, and (2) environments where the JIT is absent or restricted, such as startup-critical paths before JIT warm-up, or low-power embedded JVMs with restricted compilation budgets. In these scenarios the abstraction penalty scales linearly with depth and can compound to microsecond-level overheads per operation.

Limitations

All benchmarks in this study use monomorphic call sites. Bimorphic and megamorphic scenarios where multiple concrete types appear at the same call site, preventing devirtualization, are expected to incur additional overhead not captured here. While the `SuperChain` design simulates a non-devirtualizable call for experimental purposes, it serves only as an approximation and does not accurately reflect a truly bimorphic or megamorphic call site. Similarly, interactions with the garbage collector and object allocation patterns are not examined.

Conclusion

This paper set out to determine whether inheritance in Java imposes a quantifiable abstraction penalty, and under what conditions that penalty manifests. We examined three primary mechanisms:

1. static versus dynamic binding
2. inheritance depth

through two chain designs (Return0 and SuperChaining) executed under standard JIT, inlining-disabled, and fully interpreted JVM configurations.

Our key empirical findings are as follows. Under standard JIT, the binding penalty for dynamic over static dispatch is approximately 20% in absolute terms (roughly 0.11 ns per call). Inheritance depth has *no measurable effect* when intermediate layers perform no work due to JIT's devirtualization eliminating the cost entirely. When intermediate layers actively call `super()`, however, the JIT's inlining budget becomes the limiting factor: costs are flat up to approximately depth 8, then increase sharply, reaching a 5,840% overhead at depth 128 relative to the static baseline. With inlining disabled the scaling is near-linear with depth, and under the interpreter it is almost perfectly linear at approximately 17 ns per additional level.

An important takeaway from this work is that the inheritance hierarchy itself is not the source of runtime cost; the cost lies in whether each level of the hierarchy contributes a live `super()` dispatch, and whether the JIT can inline those dispatches away. Developers who understand which invocation patterns trigger expensive dispatch behavior can make targeted design decisions that preserve the clarity of object-oriented architecture without sacrificing runtime performance.

Further work could extend this study to interface-based dispatch versus class-based dispatch, a quantification of the performance impact of megamorphic call sites, and investigations using alternative JVM implementations such as GraalVM native image (ahead-of-time compilation) to assess how much of the abstraction penalty is JIT-specific rather than inherent to Java's object model.

References

Bogosavljević, I. 2022. Process polymorphic classes in lightning speed.

Choi, K. H. T.; and Tempero, E. 2007. Dynamic measurement of polymorphism. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, 211–220. AUS: Australian Computer Society, Inc. ISBN 1920682430.

Crisan, I. 2020. Subsystem Inheritance and Composition in Complex Systems. In *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*, 478–482.

Dean, J.; Grove, D.; and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European conference on object-oriented programming*, 77–101. Springer.

Dinesh, T. B. 1992. *Object-oriented programming: Inheritance to adoption*. Ph.D. thesis. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2025-11-06.

Kaleba, S.; Larose, O.; Jones, R.; and Marr, S. 2022. Who You Gonna Call: Analyzing the Run-Time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2022, 15–28. New York, NY, USA: Association for Computing Machinery. ISBN 9781450399081.

Kumar, P. 2022. Composition vs inheritance.

Lenovo CA. n.d. *Dynamic Binding: Enhancing Code Flexibility — Lenovo CA*.

Luján, M.; Freeman, T. L.; and Gurd, J. R. 2005. On the conditions necessary for removing abstraction penalties in OOLALA. *Concurrency and Computation: Practice and Experience*, 17(7-8): 839–866.

Misra, S.; Akman, I.; and Koyuncu, M. 2011. An inheritance complexity metric for object-oriented code: A cognitive approach. *c Indian Academy of Sciences*, 36: 317–337.

Müller, M. 2000. Abstraction Benchmarks and performance of C++ applications.

Singh, G. B. 1995. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 6(1): 30–39.

Suganuma, T.; Yasue, T.; Kawahito, M.; Komatsu, H.; and Nakatani, T. 2005. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4): 732–785.